conference

proceedings

USENIX

Proceedings of the Second Workshop on Real, Large Distributed Systems

San Francisco, CA, December 2005

WORLDS '05

Second Workshop on Real, Large Distributed Systems

San Francisco, CA, USA

December 13, 2005

Sponsored by

The USENIX Association

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $20 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$7.50 per copy for postage (via air printed matter).

ISBN 1-931971-40-4

USENIX Association

Proceedings of the Second Workshop on Real, Large Distributed Systems

December 13, 2005
San Francisco, CA, USA

Workshop Organizers

Program Co-Chairs

Brad Karp, *Intel Research Pittsburgh and Carnegie Mellon University*
Vivek Pai, *Princeton University*

Program Committee

David Andersen, *Carnegie Mellon University*
Mary Baker, *Hewlett-Packard Labs*
Mic Bowman, *Intel*
Ramesh Govindan, *University of Southern California*
Adriana Iamnitchi, *Duke University*
Dina Katabi, *Massachusetts Institute of Technology*
Eddie Kohler, *University of California, Los Angeles*
Steve Muir, *Princeton University*
Jitendra Padhye, *Microsoft Research, Redmond*
Sean Rhea, *University of California, Berkeley, and Massachusetts Institute of Technology*
Timothy Roscoe, *Intel Research, Berkeley*
Ant Rowstron, *Microsoft Research, Cambridge*
Neil Spring, *University of Maryland*

The USENIX Association Staff

External Reviewers

Fang Bian, *University of Southern California*
Min Cai, *University of Southern California*
Frank Dabek, *Massachusetts Institute of Technology*
Ramakrishna Gummadi, *University of Southern California*
Xin Li, *University of Southern California*
Mehul Shah, *Hewlett-Packard Labs*
Hui Zhang, *University of Southern California*

WORLDS '05
December 13, 2005, San Francisco, CA, USA

Tuesday, December 13, 2005

Infrastructure
Session Chair: Steve Muir, Princeton University

Choosing Wisely
Session Chair: Neil Spring, University of Maryland

Miscellaneous
Session Chair: Sean Rhea, MIT

From the Trenches
Session Chair: Mic Bowman, Intel

Index of Authors

Message from the Program Chairs

It is our pleasure to welcome you to WORLDS '05, the Second Workshop on Real, Large, Distributed Systems.

WORLDS '05 brings together people who are exploring the new challenges of building widely distributed networked systems and who lean toward the "rough consensus and running code" school of systems building. It builds on the very successful WORLDS '04, which opened a new venue for researchers to discuss a number of fresh ideas concentrated on distributed systems, including those that:

* Span a significant portion of the globe
* Spread over a large number of sites
* Are designed to run on a real platform for a period of time

In this year's papers, we see a wide range of systems—some that have grown over time and have interesting lessons for deployment, adaptations of research ideas to more hostile environments, new approaches to harness resources or optimize systems, and a number of discussions and experience papers informed by deployment and observation.

We received 28 submissions, each of which received at least three reviews, by program committee members as well as external reviewers. After careful consideration, the program committee carefully selected 12 high-quality papers for inclusion in the workshop, with the intent of providing insights into real systems, as well as fostering a lively discussion.

We would like to thank the program committee and the external reviewers for their dedication and thoroughness, without which we would not have such a strong program. We would also like to thank the USENIX staff for their professionalism, thoroughness, and flexibility, without which this workshop would not have been possible. Finally, we would like to thank the Intel Corporation for their sponsorship, which is much appreciated.

Brad Karp, University College London and Intel Research
Vivek Pai, Princeton University
WORLDS '05 Program Co-Chairs

Experience with some Principles for Building an Internet-Scale Reliable System

Mike Afergan
Akamai and MIT

Joel Wein
Akamai and Polytechnic University

Amy LaMeyer
Akamai

Abstract

We discuss the design methodology used to achieve commercial-quality reliability in the Akamai content delivery network. The network consists of 15,000+ servers in 1,100+ networks and spans 65+ countries. Despite the scale of the Akamai CDN and the unpredictable nature of the underlying Internet, we seek to build a system of extremely high reliability. We present some simple principles we use to assure high reliability, and illustrate their application. As there is some similarity-in-spirit between our implementation and recent trends in the research literature, we hope that sharing our experiences will be of value to a broad community.

1 Introduction

In this paper we discuss the design decisions made by Akamai Technologies to assure reliability. Akamai [1] provides distributed computing solutions and services, including a Content Delivery Network (CDN), which customers use to distribute their content to enhance the scalability, reliability, and/or performance of their web properties. To accomplish this, Akamai has built a network of 15,000+ servers located in 1,100+ third-party networks. On these servers we run custom software designed to ensure high reliability and performances in spite of the various failure modes of the underlying Internet.

In our context we define reliability (loosely) as the ability to deliver web content successfully to an end-user under all circumstances. The abstract goal of course is one hundred percent reliability. However, given the practical constraints of the Internet, a more practical benchmark is substantial improvement in customers' web properties reliability with Akamai as opposed to without.

Our approach to reliability is to *assume that a significant and constantly changing number of component or other failures occur at all times in the network.* Consequently, we design our software to tolerate such numerous and dynamic failures as part of the operational network. All builders of distributed systems must handle failures.

However, our decision to expect frequent and diverse failures, coupled with our widespread deployment of relatively small *regions* of machines (typically 2-18 machines), leads to a design space that is somewhat different from that of most large distributed systems. For example, in most commercial distributed systems the failure of a datacenter would be a major event, requiring immediate repair. In contrast, multiple datacenter failures in our network are not uncommon events and do not require immediate attention. In this paper, we briefly explain why we make this fundamental assumption, and then illustrate some simple design principles that are consequences of this assumption. We then demonstrate that these principles lead to a number of advantages for our software development and operations processes.

We note here what this short paper is *not.* It is not an explication of the entire Akamai architecture; we assume basic familiarity with CDN principles and Akamai functionality [6]. Nor is it an argument that the Akamai architecture is the best, nor, given the space constraints, a comparison to numerous important related efforts in distributed systems. Neither is this paper a detailed data-driven study of the reliability of our network or of the underlying components to justify our fundamental principle. Rather, this paper is born of both the sufficient maturity of our technology to enable us to identify some principles that we have applied throughout our system, and our realization that our approach resonates in many ways with recent academic activity.

One particularly relevant research effort is Recovery Oriented Computing (ROC) [5, 8]. A fundamental element of ROC's approach is component recovery, in the form of fail-stop and restart, to increase overall system reliability. In particular, "ROC emphasizes recovery from failures rather than failure-avoidance" [3]. While the approaches presented in this paper and the ROC effort developed independently, this paper can be viewed as the summary of a seven-year experiment in implementing a ROC system.

2 Our Philosophy

To begin, we first motivate and then describe our basic design philosophy.

2.1 Challenges of running an Internet-Scale System

Numerous failure modes present challenges to a would-be reliable Internet-scale system:

Path Failure: Various problems can degrade or destroy connectivity between any two endpoints in the Internet.

Machine Failure: Servers fail for a variety of reasons, from hardware failure to cable disconnects.

Region/Rack Failure: The cause here may be within Akamai's control, as when a switch fails. It may also be outside our control, such as a rack losing power, failure of an upstream router, or datacenter maintenance.

Multiple Rack/Data Center Failure: Multiple racks within a datacenter or even an entire datacenter can fail, for reasons ranging from operational error in the host network to physical problems including natural disasters.

Network Failure: ISP-internal failures can affect a subset of datacenters or even the core of a particular network.

Multi-Network/Internet-wide Failures: We continue to see issues including trans-oceanic cable cuts, Internet-scale worms (e.g., SQL-Slammer), peering problems [2], and BGP operator errors. The impact of these incidents can vary from significantly higher latency or packet loss to complete disconnection.

To better understand the nature of failures, we examine manual suspensions in our network. A manually suspended machine can communicate with the rest of the system but has been flagged to not serve traffic. The set of machines does not, for example, include servers in regions with transient packet loss—our mapping system handles those. Instead, suspended machines are those with long-term and/or habitual problems. We observe that this set has a daily churn rate of approximately 4%, supporting our assumption that components continually fail. This also implies the mean-time-to-recovery is approximately 25 days, though this distribution is heavy-tailed. For some insight into the nature of the failures, we note that approximately 40% of the suspended machines are in regions where the whole region is suspended. This is likely from a region or datacenter error.

2.2 Our Philosophy

Given the significant possibilities for failure throughout the network, we were led to the following assumption.

Our Assumption: *We assume that a significant and constantly changing number of component or other failures occur at all times in the network.*

This leads naturally to the following:

Our Development Philosophy: *Our software is designed to seamlessly work despite numerous failures as part of the operational network.*

In other words, we choose to embrace, not confront, these component failure modes. In particular, *we expect a fraction of our network to be down at all times and we design for a significant fraction of our network to be down for particular moments in time.* Losing multiple datacenters or numerous servers in a day is not unusual. While every designer of robust distributed systems expects failures and aims to provide reliability in spite of them, our philosophy, combined with our wide and diverse deployment makes our approach relatively unique among commercial distributed systems.

This philosophy pervasively informs our design. For example, we buy commodity hardware, not more reliable and expensive servers. Because the servers in a region share several potential points of failure, we build more smaller regions instead of fewer larger ones. We spread our regions among ISPs and among the datacenters within an ISP. We rely on software intelligence to leverage the heterogeneity and redundancy of our network. Finally, we communicate over the public Internet, even for internal communication. An alternative would be to purchase dedicated links; instead we augment the reliability of the public Internet with logic and software.

3 Three Design Principles

In this section we present and illustrate three practical principles that follow from our philosophy.

3.1 Principle #1: Ensure significant redundancy in all systems to facilitate failover

Realizing the principle in practice is challenging because of aspects of the Internet architecture, interactions with 3rd-party software, and cost. Because DNS plays a fundamental role in our system (it is used to direct end-users to Akamai servers), we highlight two challenges in achieving significant redundancy in DNS.

One problem is the *size constraints for DNS responses*. The Generic Top Level Domain (gTLD) servers are critical as they supply the answer to queries for `akamai.net`. However, the size of DNS responses limits the number of servers we can return to 13 [9]. This is a relatively small number and not something we can address directly. We take a number of steps to increase this redundancy, including using IP anycast.

Another problem is that *DNS TTLs* challenge reliability by fixing a resolution for a period of time. If the server fails, a user could fail to receive service until the TTL expired. We address this in two ways.

The first is a two-level DNS system to balance responsiveness with DNS latency. The top level directs the user's DNS resolver to a region with a TTL of 30 to 60 minutes for a particular domain (e.g., `g.akamai.net`).

In the typical case, this region then resolves the lower-level queries (e.g., `a1.g.akamai.net`). To prevent this single region from inducing user-appreciable failures, the top level returns (low-level) nameservers in multiple regions. This requires determining an appropriate secondary choice such that a) performance does not suffer but b) the chance of all nameservers failing simultaneously is low.

The second aspect of our approach is that, within each level, we have an appropriate failover mechanism. Because the top-level resolution is long, much of the actual mapping occurs in the low-level resolution, which has a TTL of only 20 seconds. This provides us with significant flexibility to facilitate intra-region reliability. To address failures during the 20 seconds, a live machine within the region will automatically ARP over the IP address of a down machine.

3.2 Principle #2: Use software logic to provide message reliability

The operation of Akamai's system entails the distribution of numerous messages and data within the network, including control messages. customer configurations, monitoring information, load reporting, and customer content.

One approach for these communications channels would be to build dedicated links-—or contract for virtually dedicated links—between our datacenters. We did not choose this architecture because this would not scale to a large number of datacenters. Further, since even these architectures are not immune to failures, we still would have had to invest effort in building resiliency.

We have thus built two underlying systems—a real-time (UDP) network and a reliable (HTTP/TCP) transport network. Our real-time transport network which was first built in 1999 uses multi-path routing, and at times limited retransmission, to achieve a high level of reliability without sacrificing latency. (This system is discussed in detail in [7].) We have since leveraged this system for a variety of functions including most internal network status messages.

Motivated by our customers' increasing use of dynamic or completely uncacheable content, we also built an HTTP/TCP equivalent network in 2000 to distribute content from the origin to the edge. In contrast to the UDP network and RON [4], each file is transmitted as a separate HTTP request—though often in the same connection. This HTTP subsystem serves as the basis for our SureRoute product and is an important internal communication tool. The system explores a variety of potential paths and provides an HTTP-based tunnel for requests through intermediate Akamai regions.

3.3 Principle #3: Use distributed control for coordination

This is not a surprising principle but one which we include for completeness and because there are often interesting practical subtleties. At a high level, we employ two forms of decision making. The first and most simple is failover, for cases where the logic is simple and we must immediately replace the down component. A previously discussed example of this is the case where a machine will ARP over the IP address of a down machine. The second technique is leader election, where leadership evaluation can depend on many factors including machine status, connectivity to other machines in the network, or even a preference to run the leader in a region with additional monitoring capabilities.

4 Example: How Akamai Works Under Extreme Component Failure

In this section, we examine how Principles 1-3 are applied by considering a hypothetical failure scenario.

4.1 Basic Operation

The basic HTTP request to the Akamai network consists of two parts – a DNS lookup and a HTTP request. In this section we separate the two. The DNS lookup for `a123.g.akamai.net` consists of 3 steps:

S.1) The client nameserver looks up `akamai.net` at the gTLD servers. The authorities returned are Akamai *Top Level Name Servers* (TLNSes).

S.2) The client's nameserver queries a TLNS for `g.akamai.net` and obtains a list of IP addresses of *Low Level Name Servers* (LLNSes). The LLNSes returned should be close to the requesting system. As discussed in Section 3.1, the TLNSes ordinarily return eight LLNS IPs in three different regions.

S.3) The client requests `a123.g.akamai.net` from a LLNS and obtains two server IPs.

In steps (S.1) and (S.2), the responding nameserver consults a *map* to select appropriate IPs. In the case of the TLNSes, this map is produced by a special type of region called a *Mapping Center*. Each Mapping Center runs software components that analyze system load, traffic patterns, and performance information for the Akamai network and the Internet. With this information, a top-level map is produced and distributed to the TLNSes.

When the server receives the HTTP request, it serves the file as quickly as possible subject to the configuration for the particular file. In this example we assume the file

is not on the server and the server requests the file from the origin server.

4.2 Operation under Failure

Let us now consider this operation under a case of extreme failure. Consider a model where a user selects a TLNS in network X, which also contains the current lead Mapping Center. In normal mode, the user would select Akamai region A, which would in this case need to fetch content from an origin in network C (for Customer).

We now assume three *independent and simultaneous* failures:

1. Network X has significant packet loss across its backbone.
2. Akamai region A fails.
3. Another network, B, has a problem with a peering point that is on the default (BGP) route between region $\hat{A}$ and network C.

We now examine how these problems might be handled. The particulars of the response of course depend on other circumstances and failures that may be occurring.

1. Network X degrades

- One of the metrics for leader election of Mapping Centers is connectivity to the rest of the Akamai network. Therefore, a second Mapping Center (MC) $\hat{X}$ assumes leadership from MC X and begin publishing the top-level maps. (Principles 1 and 3)
- The user's nameserver may fail over to using another one of the TLNSes. (Principle 1)
- Since some users may continue to query TLNS X, it still obtains the top level maps from MC $\hat{X}$ via the real time multi-path system. (Principle 2)

2. Region A fails

- Upon seeing that LLNS A does not respond, the client's nameserver will start using one of the other LLNSes. Let us call that LLNS $\hat{A}$. (Principle 1)
- Since region A is no longer optimal for the client, LLNS $\hat{A}$ directs the user to region $\hat{A}$. (Principle 1)
- Mapping Center $\hat{X}$ considers this failure and updates the top level maps. This includes removing region A as a choice for users and perhaps some additional load balancing. (Principle 1)

3. Network B has a problem at a peering point.

Those edge regions that use this peering point will attempt to route around this problem using SureRoute (Principle 2) by finding appropriate intermediate regions (Principle 1). This decision is made at the Edge Server in network $\hat{A}$ (Principle 3).

5 Realizing the Benefits of our Approach in our Software and Operations

In this section, we present a second group of principles relating to how Akamai designs reliable software and insulates the system from software faults. It is impossible in any system for any piece of software to function correctly always. While 100 percent correctness is our goal, we invest significant effort to ensure that if something goes wrong, the system can recover appropriately. *Most importantly, we argue that this set of principles is informed and facilitated by the first set of principles* presented in the previous two sections.

5.1 System Principle #4: Fail Cleanly and Restart

We aggressively fail and restart from a last known good state (a process we call "rolling") in response to errors that are not clearly addressable. There are two reasons that we made this decision. The first is that we knew that our network was already architected to handle server failure quickly and seamlessly. As such, the loss of one server does not pose a problem. The second is that while it may be possible to recover from these errors, the risk of operating in a mode where we are possibly behaving incorrectly can be quite high (e.g., serving the wrong homepage). Thus, comparing the low cost of rolling and the high risk of operating in a potentially corrupted state, we choose to roll.

The naive implementation as described above is however not sufficient. The problem we first encountered is that a *particular server may continually roll*. For example, a hardware problem may prevent a machine from starting up. While losing the server is not a problem, the state oscillations create problems (or at least complexity) for the overall system. Part of our current solution is to enter a "long-sleep" mode after a certain number of rolls. This is a very simple and conservative strategy, enabled by the overall system's robustness to individual machine failures.

The second problem with failing-and-recovering is *network-wide rolling*. While even a set of machines can restart at any time, it is not acceptable for the whole network (or all the machines in a particular subnetwork) to roll at once. This could be a problem for events that affect the whole network at once—such as configuration updates or a new type of request—that could suddenly trigger a latent bug. It is also a complex problem to solve. In our current solution, we enter a different mode of operation when we observe a significant fraction of the network rolling. In particular, instead of rolling we will attempt to be more aggressive in recovering from errors—and perhaps even shut down problematic software modules.

Table 1: Minimum Phase Durations

Release Type	Phase One	Phase Two
Customer Configuration	15 mins	20 mins
System Configuration	30 min	2 hours
Standard Software Release	24 hours	24 hours

5.2 System Principle #5: Zoning

In addition to standard quality practices, our software development process uses a phased rollout approach. Internally, we refer to each phase as a zone and thus call the approach *zoning*. The steps of zoning for both software and for configuration releases are:

- Phase One: After testing and QA, the release is deployed to a single machine.
- Phase Two: After monitoring this machine and running checks, the release is deployed to a single region (not including the machine used in Phase One.)
- Phase Three: Only after allowing the release to run in Phase Two and performing appropriate checks do we allow it to be deployed to the world.

The minimum duration of each phase per release type is summarized in Table 1. The actual time can be longer based on the nature of the release and other concurrent events. On the extreme end, operating system releases are broken into many more sub-phases and we will often wait days or weeks between each phase.

While the concept of a phased rollout is not unique, our system is interesting in that the process is explicitly supported by the underlying system. This facilitates a more robust and clean release platform which benefits the business. The properties of redundancy (P1) and distributed control (P3) enable us to lose even sets of regions with minimal user-appreciable impact. Consequently, this obviates any concern about taking down a machine for an install—or even a full region in Phase Two. This is in stark contrast even to some well-distributed networks. For example, a system with 5 datacenters is more distributed that most, but even in this case, taking even one datacenter down for an install reduces capacity by a 20%—significant fraction.

We believe this is an example where our design principles present us with a significant yet unexpected benefit. Principles 1 (significant redundancy), 3 (distributed control), and 4 (aggressively fail-stop) were not chosen to facilitate software roll-outs. However, over time we have seen that these principles have enabled a much more reliable and aggressive release process, both of which have been a huge benefit to our business. We present some metrics substantiate this claim in Section 6.1.

5.3 System Principle #6: The network should notice and quarantine faults

While many faults are localized, some faults are able to move or spread within the system. This is particularly true in a recovery-oriented system—and this behavior must be limited. One hypothetical example is a request for certain customer content served with a rare set of configuration parameters that triggers a latent bug. Simply rolling the servers experiencing problems would not solve the problem since the problematic request would be directed to another server and thus spread. It is important that such an event not be allowed to affect other customers to the extent possible.

We address this problem through our our low-level load balancing system. Low level load balancing decides, for each region, to which edge servers to assign which customer content. To achieve fault isolation, we can constrain the assignment of content to servers to limit the spread of particular content. In the unlikely case that some customer content instantly crashes all the servers to which it is assigned, this approach allows us to mostly serve all other content.

To effectively respond when seeing problems, we must involve both localized and globally distributed logic. In a purely local solution, we could constrain all regions equally. This has the downside that, for example, a customer with geographically localized traffic may produce more load relative to the number of servers than we'd be able to load balance effectively, even though the customer posed little risk to the global network. As a result, we've designed a two-level solution. The mapping system, via the messaging infrastructure, gathers data from all regions and tells the low-level load balancing systems in each region what constraints they need to impose in order to keep the system in a globally safe state.

6 Evaluation

Evaluating our decisions in a completely scientific fashion is difficult. Building alternative solutions is often infeasible and the metric space is multi-dimensional or ill-defined. While it is not a rigorous data-driven evaluation, this section presents two relevant operational metrics.

6.1 Benefits to Software Development

Zoning, as presented in Section 5.2, allows us to be more aggressive in releasing software and other changes by

Table 2: Software Release Abort Metrics

Phase	Number Aborted	Percent of Total Releases
Phase One	36	6.49%
Phase Two	17	3.06
Add'l Phase	3	0.54
World	23	4.14

exploiting our resilience to failure. Over the past year, we have averaged 22 software and network configuration releases and approximately 1000 customer configuration changes (approximately 350 change bundles) per month. Here we examine the number of aborts in each phase of the release process as a metric for the number of errors found. This metric is somewhat subjective. It depends on our ability to catch the errors and our willingness to allow a release with a known (presumably small) problem to continue to a subsequent phase. Further, the optimal value for this metric is not clear. Zero aborts is an obvious goal; however, seeing zero aborts would likely imply little trust in the network's resiliency—and likely reflect a longer time-to-market for features. On the other hand, too many aborts would suggest poor software engineering practices. An ideal network would likely have some, but not too many, aborted releases.

Table 2 presents data taken from several hundred software and network configuration releases that occurred between 7/1/02 and 8/9/04 (25 months). Despite the aforementioned limitations, several observations can be made. First note that the overall level of aborts is roughly as we desire, relatively low but not zero, at 14.23%. Second, when we do abort a release it is most often in Phase One, where the impact to the network is extremely minor. We also see, surprisingly so, that the number of aborts in the World Deploy phase is greater than the number of aborts in Phase Two. This is due in part to the complexity and difficulty in testing a large-scale real-world system. (This is an area of ongoing research at Akamai and we believe an emerging research area of broader interest.) Finally, we note that all the all of the problems found in zoning made it through substantial developer testing and QA.

6.2 Benefits to Operations

A third interesting metric is the number of employees and the amount of effort required to maintain the network. There are many factors that could make our network difficult to maintain—including the size, the large number of network partners, the heterogeneity of environments, and less technical factors such as long geographical distances, different timezones, and different languages.

However, these challenges are mitigated by our fundamental assumption. In particular:

- We assume in our design that components will fail. Therefore, the NOCC does not need to be concerned by most failures.
- Since a component (e.g. machine, rack, datacenter, network) failure does not significantly impact the network, the system and the NOCC can be aggressive in suspending and not using components. Even if the NOCC is mildly concerned, it can suspend a component.
- We assume that a fraction of components will be down at any time. Therefore the NOCC does not need to scramble to get the components back online.

As a result, our minute-to-minute operations team can be quite small relative to the size of our network. In particular, our NOCC has seven or eight people on hand during the day and only three people overnight. For the sake of comparison, that means a ratio of over 1800 servers per NOCC worker during the day and over 5000 servers per NOCC worker at night.

Acknowledgments

The systems discussed in this paper were designed and developed by numerous Akamai employees past and present. We are grateful to Armando Fox for a very careful reading of an early version of the paper and many useful comments and the help of our shepard, Brad Karp. We also gratefully acknowledge conversations with Steve Bauer, Rob Beverly, Leonidas Kontothonasis, Nate Kushman, Dan Stodolsky, and John Wroclawski.

References

[1] Akamai technologies homepage. http://www.akmai.com/.

[2] Net Blackout Marks Web's Achilles Heel. http://news.com.com/2100-1033-267943.html?legacy=cnet.

[3] Recovery-Oriented Computing: Overview. http://roc.cs.berkeley.edu/roc_overview.html.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[5] A. Fox. Toward Recovery-Oriented Computing. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, 2002.

[6] J. Dilley et al. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[7] L Kontothanassis et al. A Transport Layer for Live Streaming in a Content Delivery Network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[8] D. A. Patterson. Recovery Oriented Computing: A New Research Agenda for a New Century. In *HPCA*, page 247, 2002.

[9] I. E. Paul Vixie and W. Akira Kato. Dns response size issues. Dnsop working group ietf internet draft, July 2004. http://www.ietf.org/internet-drafts/draft-ietf-dnsop-respsize-01.txt.

Deploying Virtual Machines as Sandboxes for the Grid

Sriya Santhanam*, Pradheep Elango, Andrea Arpaci-Dusseau, Miron Livny
University of Wisconsin-Madison
{sriya, pradheep, dusseau, miron}@cs.wisc.edu

Abstract

The ability to securely run arbitrary untrusted code on a wide variety of execution platforms is a challenging problem in the Grid community. One way to achieve this is to run the code inside a contained, isolated environment, namely a "sandbox". Virtual machines provide a natural solution to the security and resource management issues that arise in sandboxing. We explore different designs for the VM-enabled sandbox and evaluate them with respect to various factors like structure, security guarantees, user convenience, feasibility and overheads in one such grid environment. Our experiments indicate that the use of on-demand VMs imposes a constant startup overhead, with I/O-intensive applications incurring additional overheads depending on the design of the sandbox.

1 Introduction

Security vulnerabilities in grids have been the focus of a lot of attention in recent years. Butt et al. [4] discuss the inadequacy of traditional safety checks, while Miller et al. [13] emphasize the susceptibility of current grid systems by demonstrating a security exploit on a grid using dynamic code instrumentation techniques. Though such specific problems can be tackled, they indicate the dangers of grid applications leaving contaminating residual state. Further, common attacks such as root exploits and stack smashing can always bring down a machine and are a major worry for widely-used grid environments like Condor [8].

To address these concerns, a grid computing system must enable the execution of untrusted, unverified applications within a secure, isolated environment, namely a "sandbox". Essentially, the impact of the running application must be restricted to the sandbox, thereby protecting the underlying host machine. Depending on the application's requirements, access to resources on the host machine can be selectively allowed from within the sandbox. Moreover, while security attacks and faults can occur within the sandbox, its framework must guarantee that these vulnerabilities do not affect the underlying execute machine.

In this paper, we explore the use of Virtual Machine (VM) technology to implement sandboxes. VMs present the image of a dedicated raw machine to the grid application. An application running on a VM is decoupled from the system software of the underlying host machine. This ensures that an untrusted user or application can only compromise the VM and not the underlying physical resource. VMs also enable fine-grained resource allocation for grid jobs. It is hence feasible to restrict the memory, network, disk size, and even the CPU cycles allocated to a given VM. Furthermore, the use of VMs allows the target execution environment for a grid application to be completely customized, thereby enabling support for jobs with special requirements like root access or legacy dependencies. VMs also enable process migration without requiring any modification or relinking of the grid application.

As a result, considerable attention has been focused on the idea of integrating VMs into the grid architecture [5, 9, 11, 19]. However, the notion of the "sandbox" that these VMs are used to create is not uniformly defined, particularly with respect to its structure. A sandbox is defined both by what it contains and where its boundaries are. VMs offer the flexibility to implement different sandbox designs and in this paper, we attempt to explore and evaluate various implementations of the VM-based sandbox along factors such as structure, security guarantees, feasibility, user convenience and performance overheads.

Here we present four different sandbox designs, ranging from a simple substitution of a grid node with a virtual node all the way up to on-demand VMs that use lazy file-retrieval techniques. The choice of sandbox design, as we emphasize in this effort, is workload dependent and our aim is to give a clear understanding of the pros and cons of each solution. We chose to implement our sandboxes on Condor [12], a large, distributed grid environment that is currently deployed across the globe more than 1700 Condor pools, representing

*Author is currently working at VMware, Inc. This work was done while the author was studying at the University of Wisconsin-Madison.

over 60,000 computers. Our sandboxes are deployed using Xen [3] as the virtual machine monitor, which in contrast to other virtual machine monitors that support full virtualization, has been proven to have near-native Linux performance with very low overheads.

2 Current Approaches

VMs are not the only solutions to the problem of sandboxing. Simpler abstractions such as chroot() have been used for the same purpose. However, such sandboxes are limited as they can isolate only the filesystem parts, and further, simple exploits for breaking chroot() are well-known [7]. Programming language virtual machines such as the Java Virtual Machine support similar goals of sandboxing and portability. However they severely restrict the range of applications that can run on them. Jails [10] and Pods [14] are other abstractions that could provide sandboxing. The jail mechanism is common in the FreeBSD world, and is a stronger variant of chroot() sandboxes. Jails, however, do not provide much support for other desirable properties such as migration. Pea-pods [15], an extension of Pods, support migration although this migration is restricted to different versions of the same kernel. While adapting such mechanisms for the grid environment could open up interesting research avenues, we choose to work with VMs owing to the several benefits they provide along with the flexibility they offer for different sandboxing solutions.

The idea of integrating VMs with grids was initially popularized by Figueriredo et al.[9], whose observations subsequently led to the INVIGO project[1]. However, the INVIGO system addresses higher level issues such as service discovery and composition, paying relatively lesser attention to lower-level issues like security and isolation. Further, INVIGO uses a VM monitor that supports full virtualization, while we use a paravirtualized monitor which would incur much lesser overheads. Entropia [5] and SETI@Home[16] address the issue of sandboxing using binary rewriting solutions that only support custom-developed grid applications; thus they restrict the range of applications that can run on their grid.

3 Design and Implementation

3.1 Sandbox 1: Virtual grid nodes

This is the simplest sandbox in terms of design and implementation. A virtual machine joins the grid just like any other physical machine. It serves as a substitute for the underlying physical machine in the grid.

Even this simple design however offers most of the benefits of VM technology, as summarized by Table 1. For instance, a node can be easily configured to control resource allocation for the grid applications, restricting their use of main memory, disk usage, swap space and CPU cycles. Further, since all resources are virtualized on this node, its configuration can be different from that on the physical node; for example, the network card on the VM could be restricted by firewall policies that are entirely different from that on the physical machine. Virtual nodes also provide isolation from the other applications that are running on the physical machine. Isolation can prevent attacks from residual processes of a grid job that could have run previously, in addition to preventing security attacks and faults from spreading over to the physical machine.

In our implementation, we replace the execute machine that Condor sees with a Xen VM. The VM has network access and is typically assumed to be running in the background on the execute machine during normal operation. Condor is installed only on the VM and not on the underlying physical machine. This ensures that jobs run only inside the VM and are never directly executed on the physical machine. The only drawback of this design is that the application is still open to the network, and can hence launch network attacks on any accessible system.

3.2 Sandbox 2: Eager prefetching, whole-file-caching sandboxes

In this design, VMs act as individual job containers with no network access. To guarantee correctness while disallowing network access, the VM must contain all job requirements within the sandbox before the job begins execution. It does this by eagerly fetching all the data files needed by the job prior to execute time. All I/O requests made by the job at run time will hence be satisfied locally within the sandbox.

This design uses VMs as ad-hoc entities that exist only for the duration of the job. While this incurs the overheads of starting and shutting down a virtual machine for each run of a grid job, on the flip side, system resources are consumed only for the job duration. This solution also allows the flexibility of launching a VM with a pre-configured environment that matches the job's requirements, although since job dependencies have to be identified by the user beforehand, there is a downside on user convenience. However, for applications with well-defined I/O behavior, this design provides the tightest guarantees of sandboxing.

3.3 Sandbox 3: Lazy, block-caching sandboxes

In this design, the boundary of the sandbox is extended to include the original machine from which

the job was submitted, i.e. the "submit machine". System calls are trapped and executed remotely at the submit machine; only the results are transferred back to the execute machine. This avoids any library or software compatibility issues as the applications are not tied to the software configuration of the underlying physical system.

This solution will be particularly useful when an application has huge input file dependencies but makes few I/O calls. As before, VMs are launched on-demand. User convenience is enhanced since the user need no longer specify job dependencies. However, since the VM is configured with restricted network access, limited network attacks are still possible. The security guarantees provided by this sandbox are hence not as tight as those provided by Sandbox 2. Sandbox 4 hence attempts to minimize this tradeoff and bring about the best of both worlds.

3.4 Sandbox 4: Lazy prefetching, whole-file caching sandboxes

This design is very similar to our second sandbox design with the main difference being that the decision to prefetch the files required by the job is done dynamically at job run time rather than statically prior to the execution of the job. Hence, this sandbox only transfers entire files when it sees that a file being opened exists on the submit machine. However, in order to transfer data over the network, we need to open up network access to the sandbox, which in turn creates an opportunity for the job to attack/be attacked on the network. Hence, to prevent network attacks in this scenario, the sandbox must suspend the executing job for the time period during which the network is opened.

Our implementation uses Chirp [17], a lightweight system for performing file I/O over a network. Chirp allows file access to be set up with fine-grained control so that user permissions are not violated. Another component of this implementation is Bypass, an application that is used to create interposition agents and split execution systems. This implementation launches a VM that has no network access. All open system calls from the Condor job are interposed. An open call that requests I/O on the network, forces the job to be suspended and a restricted network connection to be opened up. The requested file is then copied to the local machine. Network access is then disabled, and the request proceeds normally. A log maintains new changes to the directory structure, which are merged onto the submit machine after the job completes.

User convenience and flexibility in this scenario are achieved because the user does not have to specify any dependencies for the job. Hence this solution servers as a tradeoff between between designs 2 and 3, offering the tighter security guarantees of sandbox 2 while maintaining the degree of user convenience offered by sandbox 3.

3.5 Suspend/Resume

For all the on-demand VM solutions, it is also possible to preserve process state when the Condor job needs to be vacated from the execute machine. Under normal conditions, Condor jobs are vacated from an execute machine once the machine detects keyboard activity and ceases to be idle. However, since jobs are now running inside a VM, it is possible to suspend and resume the VM on a different machine rather than lose the progress made by the job by restarting it on another machine. This mechanism, already supported by the virtual machine monitor, helps preserve job state across different machines.

To implement this functionality, the wrapper application sets up a signal handler to trap the SIGTERM signal, which is the signal sent by the Condor daemon when it wants to vacate a job. The wrapper then suspends the VM state to a restore file on receipt of this signal, and subsequently exits. Condors file transfer mechanisms are set up to transfer all files written by the job on eviction or exit from the execute machine. Hence, the next time the job is scheduled to run on a (possibly different) machine, the wrapper checks for the presence of the restore file, and if present, resumes the VM rather than booting it up again. The Condor job continues seamlessly from the point of eviction.

4 Evaluation

We characterize the different sandbox designs with respect to factors such as security guarantees, application functionality and user convenience in Table 1. For our experiments, we have chosen to focus on data-centric micro-benchmarks, since applications of this category form a major share of grid workloads [2, 18].

Further, in a virtualized environment like Xen, data-intensive workloads entail a significant CPU overhead [6] and are hence likely to be most affected by our sandboxing techniques. Computation-intensive workloads on the other hand, on account of their minimal interaction with the OS, tend to perform competitively with negligible overhead on the Xen VMM [3]. Data-centric workloads could vary in their characteristics - in the number of files accessed, amount of data transferred as well as the burstiness of I/O rates [18]. We present the results of the experiments that highlights the performance overheads and scalability of

Feature	Condor Vanilla Universe [1]	Condor Standard Universe [1]	Sandbox 1	Sandbox 2	Sandbox 3	Sandbox 4
Security Guarantees						
Protecting execute machine from malicious/faulty job	No	No	Yes	Yes	Yes	Yes
Preventing network attacks by a malicious job	No	No	No	Yes	Limited	Yes
Protecting against lurker processes (one Condor job attacking another)	No	No	No	Yes	Yes	Yes
Application Functionality						
Customizing job execution environment (Libraries, Packages, etc)	Limited	Limited	Limited	Yes	Yes	Yes
Running jobs with network requirements	Yes	Yes	Yes	No	Limited	No
Running jobs which require higher privileges (e.g. root access)	No	No	Yes	Yes	Yes	Yes
Running jobs which require a different OS than the one on the execute machine	No	No	Yes	Yes	Yes	Yes
Reflects all writes made by the job	No	Yes	Only in Standard Universe	No	Yes	Yes
Administrator and End-user Usability						
Controlling resource allocation (CPU, memory, disk) on execute machine	Limited	Limited	Yes	Yes	Yes	Yes
Job Migration	No	Yes	Only in Standard Universe	Yes	Yes	Yes
Requires user to specify job requirements	Yes	No	Only in Vanilla Universe	Yes	No	No

Table 1: *Comparison of the four sandboxes with respect to various metrics*

the different designs below.

4.1 Platform

To implement our sandboxes, we used Xen 2.0 on Linux 2.6.9 kernels which run Condor 6.7.6. Condor [12] is a large, distributed grid environment that is currently deployed across the globe more than 1700 Condor pools, representing over 60,000 computers. All our experiments were performed on Dell workstations: 2.6GHz Xeon processors, each using a 120GB Seagate IDE hard disk (7200-RPM, 8.5 ms avg seek time). While a few of the experiments were conducted using the virtual network provided by Xen, in order to take real network overheads into account, we conducted certain experiments over a Gigabit ethernet. The VMs were configured with 128 MB memory and 2GB (virtual) hard disk capacity.

[1] Condor allows two types of jobs, "standard" and "vanilla". Standard jobs can be checkpointed and migrated from system to system transparently by Condor without restarting. However, for a code to be submitted as a standard job it must be recompiled using various Condor-specific compiler options and libraries.

4.2 Experiment 1: A comparison of the four sandboxes

In this experiment, we ran jobs performing 40 million random I/O calls over a physical network, on each of the sandbox designs. Figure 1 shows that Sandbox 1 performs equivalently with Sandbox 2 when jobs are submitted over the network. The worst performing sandbox is Sandbox 3, on account of all 40 million calls being issued remotely. The additional run time overhead incurred by Sandbox 4 is on account of the network transfer delay for the input file. Queuing delays are the highest for Sandbox 3 on account of its implementation requiring three job submissions. For the remaining experiments, we focus our comparison on jobs running on Condor Standard Universe, Sandbox 3 and Sandbox 4 since it is only in these environments that the job runtime involves network transfer overheads that are comparably affected by the amount of I/O performed by the job. In Condor Vanilla Universe, Sandbox 1 and Sandbox2, all the files required by the job are transferred prior to job execution and all I/O is performed locally; job runtimes in these environments are hence not affected

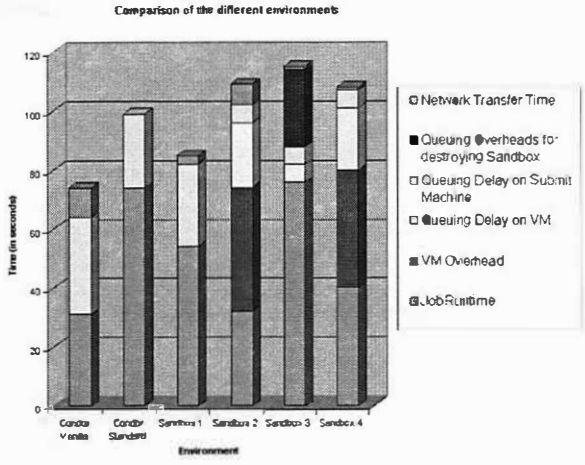


Figure 1: **Experiment 1.** *A comparison of execution times of jobs on the different sandboxes. Condor Vanilla and Condor Standard are the base Condor environments without sandboxing.*

materially by variations in I/O patterns.

4.3 Experiment 2: Varying the input file size

In this experiment, we vary the size of the input file required by a job and demonstrate the effects. Each experiment issues 10,000 reads on an input file the size of which has been varied from 4K to 256M. All reads are random reads, so prefetching and caching effects are factored out. Sandbox 3 and Standard universe jobs transfer all I/O requests over the network. Network overheads are a little higher in Sandbox 3 due to virtualization effects and our measures to guarantee security. Figure 2 shows that both Sandbox 3 and Standard universe jobs follow similar trends. While there is a fixed cost on Sandbox 3 jobs, Sandbox 4 transfers the files during run time. The above graph shows that Sandbox 4 can be a really useful option for I/O intensive jobs. The huge hit in the run time for a 256 MB file is on account of thrashing effects as our VM memory size was limited to 128 MB. The first two sandbox designs are not included in this comparison because the job runtime will not factor in the transfer time for the file.

4.4 Experiment 3: Varying the number of input files

In this experiment, we vary the number of input files required by a job. On every file, we issue 10,000 random read calls. The size of each file was 4 MB. Figure 3 shows that sandbox 3 and the standard universe scale smoothly since they only transfer individual I/O requests instead of entire files.

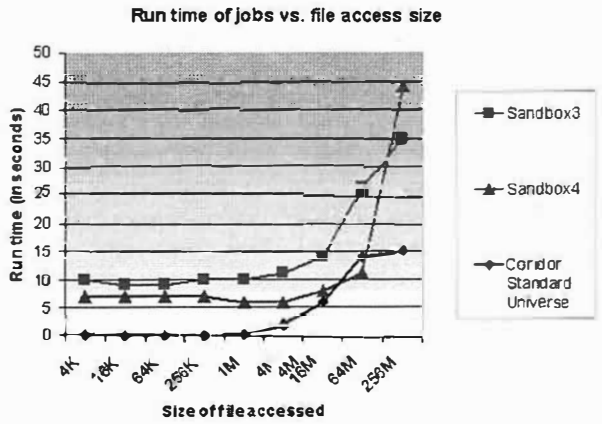


Figure 2: **Experiment 2.** *The effect of input file size on run times.*

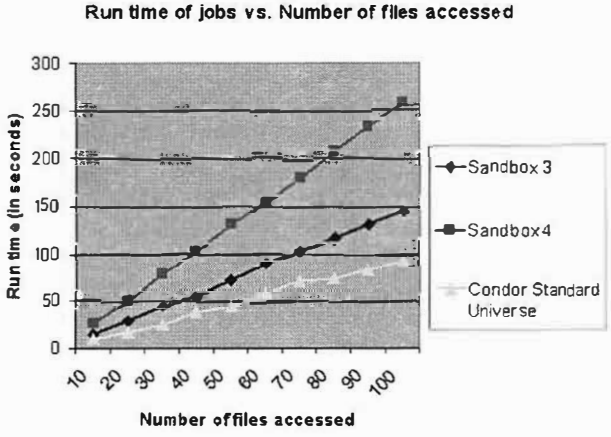


Figure 3: **Experiment 3.** *Run time vs Number of files accessed.*

Sandbox 4 however scales a lot more steeply, since it fetches whole files from the submit machine on each open call. Sandbox 3 hence favors scenarios which involve a lot of input files with relatively fewer I/O operations.

4.5 Overheads of Suspend/Resume

Our implementation of process migration for ad-hoc VM-based sandboxes was built using Xen's suspend/resume capabilities. In our experiments, on average it took 5 seconds for a Xen VM configured with 128 MB of RAM to suspend and about 7 seconds for the same VM to be restored. For this configuration, the size of the suspend file created by Xen and transferred by Condor was 92 MB, which took on average 15 seconds to transfer over the network. VM disk images were cached on the execute machine and hence only the sus-

pend file was transferred across the network. These figures remain static across multiple workloads; a short-lived job running inside the VM will take the same amount of delay to suspend as compared to a long-running application with several pending I/O requests. In general, whenever process migration is possible using mechanisms such as Condor's checkpointing for its Standard Universe jobs, it should be preferred over VM migration since VM migration is a more heavy weight operation. However, this provides a useful alternative for applications that cannot otherwise be checkpointed.

5 Conclusions

Virtual machines prove to be a natural platform for sandboxing in grids, offering a host of benefits like fine-grained resource control and allocation, fault isolation, customized execution environments, and support for process migration among others. The downside of using virtual machines for this purpose is the performance overhead characterized by our experiments. However, given that applications submitted to run in a grid environment like Condor must tolerate delays on the order of several minutes, we believe that these overheads should be acceptable by users who desire the benefits of this approach.

Table 1 highlights and contrasts the features of the different sandbox designs. As indicated, each design offers different levels of security and user convenience. One interesting challenge is the ability to support jobs requiring legitimate network access in a sandbox, while protecting all the hosts on the network. Although we have explored four different styles of sandboxes in this effort, VMs offer very flexible solutions to the problem of sandboxing and hence there are other possible designs that may suit different grid environments. Among our chosen implementations, Sandbox 4 offers the most in terms of flexibility but takes a hit in performance if the workload is I/O intensive. As a limited yet simple solution to sandboxing, Sandbox 1 offers an easy alternative as well.

In the future, we expect to see virtual machines as first class objects integrated into core grid architectures for purposes of sandboxing.

References

[1] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu. From Virtualized Resources to Virtual Computing Grids: The In-VIGO System, Generation Computing Systems, Special Issue, Complex Problem-Solving Environments for Grid Computing, David Walker and Elias Houstis, Editors, June 2005.

[2] G. Allen, T. Goodale, M. Russell, E. Seidel, and J. Shalf. Classifying and Enabling Grid Applications. In *Berman, F., Fox, G.C., Hey, A.J.G. (Eds.); Grid Computing, Making the Global Infrastructure a Reality. Wiley, Chapter 23*, 2003.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[4] A. Butt. Grid-computing Portals and Security Issues. In *Journal of Parallel and Distributed Computing, Academic Press, Inc. Orlando, FL, USA, 2003*, 2003.

[5] B. Calder, A. A. Chien, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. In *Journal of Parallel and Distributed Computing, Volume 63(5), pp 597-610*, 2003.

[6] L. Cherkasova and R. Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.

[7] How to break out of a chroot() jail. http://www.bpfh.net/simes/computing/chroot-break.html .

[8] Bruce Beckles, Security Concerns with Condor: A brief overview, http://www.nesc.ac.uk/esi/events/438/ security_concerns_overview.pdf.

[9] R. J. Figueiredo, P. A. Dinda, and J. A. Fortes. A Case For Grid Computing on Virtual Machines. In *Proceedings of the International Conference on Distributed Computing Systems(ICSDS)*, May 2003.

[10] P. H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *The FreeBSD Project, http://phk.freebsd.dk/pubs/sane2000-jail.pdf*, 2000.

[11] K. D. Katarzyna Keahey and I. Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, November 2004.

[12] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In 8th International Conference on Distributed Computing Systems, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press, 1988.

[13] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A.Mirgorodskii, and F. Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes, parallel process. lett. 11 (2,3) (2001) 267 - 280, 2001.

[14] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

[15] S. Potter, J. Nieh, and D. Subhraveti. Secure Isolation and Migration of Untrusted Legacy Applications. In *Columbia University Technical Report CUCS-005-04*, January 2004.

[16] SETI Institute, Online, http://www.seti-inst.edu/science/setiathome html.

[17] D. Thain. Chirp User's Manual. http://www.cse.nd.edu/ ccl/software/manuals/chirp.html, 2004.

[18] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and Batch Sharing in Grid Workloads. In *Proceedings of High-Performance Distributed Computing (HPDC-12)*, page 152161, June 2003.

[19] M. Zhao, J. Zhang, and R. J. Figueiredo. Distributed File System Support for Virtual Machines in Grid Computing. In *Proceedings of the High Performance Distributed Computing (HPDC)*, July 2004.

MON: On-demand Overlays for Distributed System Management *

Jin Liang, Steven Y. Ko, Indranil Gupta and Klara Nahrstedt
University of Illinois at Urbana-Champaign
{*jinliang, sko, indy, klara*}*@cs.uiuc.edu*

Abstract

This paper presents the management overlay network (MON) system that we are building and running on the PlanetLab testbed. MON is a distributed system designed to facilitate the management of large distributed applications. Toward this goal, MON builds *on-demand* overlay structures that allow users to execute *instant management commands*, such as query the current status of the application, or push software updates to all the nodes. The on-demand approach enables MON to be light-weight, requiring minimum amount of resources when no commands are executed. It also frees MON from complex failure repair mechanisms, since no overlay structure is maintained for a prolonged time. MON is currently running on more than 300 nodes on the PlanetLab. Our initial experiments on the PlanetLab show that MON has good performance, both in terms of command response time and achieved bandwidth for software push.

1 Introduction

In recent years, large distributed computing systems such as the PlanetLab [18] are increasingly being used by researchers to experiment with real world, large scale applications, including media streaming, content distribution, and DHT based applications. While a realistic environment like the PlanetLab can often provide valuable insights lacking in simulations, running an application on it has been a difficult task, due to the large scale of the system, and the various kinds of failures that can occur fairly often [8]. Thus an important tool is needed that helps application developers to *manage* their applications on such systems.

Imagine a researcher who wants to test a new application on the PlanetLab. To do this, the researcher needs to first *push* the application code to a set of selected nodes, then *start* the application on all the nodes. Once the application is running, the researcher may want to *query* the current status of the application, for example, whether the application has crashed on some nodes, and whether some error message has been printed out. Later, if a bug is identified, the researcher may want to stop the application, upload the corrected version, and start it again. To accomplish the above tasks, what the researcher needs is the ability to execute some *instant management commands* on all the selected nodes, and get the results immediately. Although many useful tools exist on the PlanetLab, such as status monitoring and query [1, 15, 4, 10, 11], resource discovery [16], and software distribution [17, 6], few of them allow users to execute instant management commands pertaining to their own applications.

PSSH [5] and vxargs [7] are two tools for executing commands on large number of machines in parallel. However, both tools use a centralized approach, where each remote machine is directly contacted by a local process. This may have scalability problems when the system becomes large, or when large amount of data needs to be transferred. The centralized approach also means there is no in-network aggregation. Thus all the execution results are returned to the local machine, even though only their aggregates are of interest.

In this paper we present the management overlay network (MON) system that we are building and running on the PlanetLab. MON facilitates the management of large distributed applications by allowing users to execute instant management commands pertaining to their applications. For scalability, MON adopts a distributed management approach. An overlay structure (e.g., a spanning tree) is used for propagating the commands to all the nodes, and for aggregating the results back.

Maintaining an overlay structure for a long time is difficult, due to the various kinds of failures that can occur in a large system. For example, if a tree overlay has

*This work was in part supported by NSF ANI grant 03-23434, NSF CAREER grant CNS-0448246 and NSF ITR grant CMS-0427089.

been created and some interior node has crashed, the tree structure must be repaired by the disconnected nodes rejoining the tree. The rejoin could become very complex, if multiple nodes fail at the same time. As a result, MON takes an *on-demand* approach. Each time a user wants to execute one or more management commands (called a *management session*), an overlay structure is dynamically created for the commands. Once the commands are finished, the overlay structure is discarded. This on-demand approach has several advantages. First, the system is simple and lightweight, since no overlay structure is maintained when no commands are executed. Second, on-demand overlays are likely to have good performance, since they are built based on the current network conditions. Long-running overlays, even if they can be correctly maintained, may have degraded performance over time. Third, since the overlays are created on-demand, different structures can be created for different tasks. For example, trees for status queries and DAGs (directed acyclic graphs) for software push.

MON is currently running on more than 300 nodes on the PlanetLab, and supports both status query (e.g., the aggregate information of different resources, the list of nodes that satisfy certain conditions, etc.) and software push commands. Our initial experiments show that MON has good performance. For a simple status query on more than 300 nodes, MON can propagate the command to all the nodes and get the results back in about 1.3 seconds on average. For a software push to 20 nodes, MON can achieve an aggregate bandwidth that is several times that an individual node can get from our local machine.

In the rest of the paper, we first present the architecture and design of MON in Section 2, then provide our evaluation results in Section 3. Section 4 provides more discussion about MON and Section 5 is the conclusion.

2 MON Architecture and Design

The MON system consists of a daemon process (called a MON server) running on each node of the distributed system. Each MON server has a three layer architecture as shown in Figure 1. The bottom layer is responsible for membership management. The middle layer is responsible for creating overlays (e.g., trees and DAGs) on-demand, using the membership information from the bottom layer. Once an overlay structure has been created, the top layer is responsible for propagating management commands down to the nodes, and aggregating the results back.

2.1 Distributed membership management

Maintaining up-to-date global membership for a large distributed system is difficult, especially when nodes

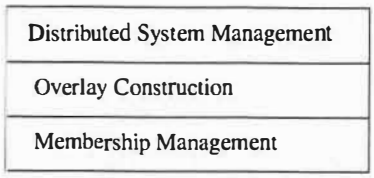


Figure 1: MON Architecture

can fail and recover fairly often. As a result, we adopt a gossip-style membership management. Specifically, each node maintains a partial list of the nodes in the system (called a *partial view*). Periodically, a node picks a random target from its partial view, and sends a `Ping` message to it, together with a small number of membership entries randomly selected from the partial view. A node receiving a `Ping` message will respond with a `Pong` message, which also includes some random membership entries. The `Ping` and `Pong` messages allow the nodes to learn about new nodes and to detect node failures. They also allow nodes to estimate the delay between each other. Such delay information can be used by the middle layer to construct locality aware overlays.

In order to maintain the freshness of membership entries, each entry is associated with an *age*, which estimates the time since a message is last received from the corresponding node. When the partial view is full and some entries need to be dropped, the oldest entries are dropped first.

2.2 On-demand overlay construction

On-demand overlay construction is a central component of our MON system. In this paper we consider the construction of two kinds of overlay structures, trees and directed acyclic graphs (DAGs). A tree structure is suited for instant status query, and a DAG is suited for software push. Since an overlay is created on-demand, we would like the construction algorithm to be quick and efficient, involving minimum startup delay and message overhead.

Ideally one may want to create an overlay that includes all the current live nodes (i.e., has full *coverage*). However, "all current live nodes" is a slippery term in a large dynamic system. In fact, merely counting the number of such nodes is a difficult task [14]. As a result, we are content with probabilistic node coverage and focus on quick and efficient overlay construction algorithms [1].

Tree Construction. The first algorithm we consider is *random tree construction*. To create an on-demand overlay tree, a client side software (called a MON client) sends a `Session` message to a nearby MON server. Each node (MON server) that receives a `Session` message for the first time will respond with a `SessionOK` message and become a child of the `Session` sender. It

also randomly picks k nodes from its partial view, and send the `Session` message to these nodes. k is called the fanout of the overlay and is specified in the `Session` message. If a node receives a `Session` message for a second time, it will respond with a `Prune` message. It has been shown that assuming the partial views represent uniform sampling of the system, such tree construction will cover all the nodes with high probability, if $k = \Omega(\log N)$, where N is the total number of nodes in the system [12].

The random tree construction algorithm is simple and has good coverage (with sufficient fanout k). However, it is not locality aware. Therefore we have designed a second algorithm called *two stage*, which attempts to improve the locality of a tree, while still achieve high coverage. To do this, the membership layer of each node is augmented with a *local list* in addition to the partial view, which consists of nodes that are close by. Each node is also assigned a random *node id*, and the local list is divided into *left* and *right* neighbors (those with smaller and larger node ids).

The tree construction is divided into two stages. During the first several hops, each node selects its children randomly from the partial view, just like the random algorithm. The goal is to quickly spread the `Session` message to different areas of the network. In the second stage, each node first selects nodes from its local list, then from the partial view if not enough local neighbors are present. To prevent nearby nodes from mutually selecting each other as children, equal number of children are selected from the left and right neighbors.

DAG construction The above tree construction algorithms can be modified to create DAGs (directed acyclic graphs). Specifically, each node is assigned a *level* l. The level of the root node is set to 1. The level of a non-root node is 1 plus the level of its first parent. Suppose a node has set its level to l and a second `Session` message is received, it can accept the sender as an additional parent, as long as its level is smaller than l. This ensures the resulting overlay contains no loop, thus a DAG.

2.3 Instant command execution

Once an overlay structure is dynamically created, one or more management commands can be executed on it. We discuss two types of management commands: status query and software push.

Status Query All the status query commands are executed in a similar fashion. First the command is propagated down the overlay tree to all the nodes. Next the command is executed locally on each node. Finally the results from the children nodes and from the local execution are aggregated and returned to the parent. Below is a (partial) list of the status query commands that we have implemented.

- `count`
- `depth`
- `topology`
- `avg <resource>`
- `top <num> <resource>`
- `histo <resource>`
- `filter <operation>`

The first three commands return information about the overlay itself, such as the number of nodes covered, the depth of the tree/DAG, and the topology of the overlay. The next three commands return the aggregate information (e.g., average, top k, and histogram) of different resource. Currently MON supports resources such as the CPU load, free memory, disk usage, number of slices, etc. Most of these resources are obtained from the CoTop [2] server on each PlanetLab node.

The last command allows (in theory) any arbitrary operation to be executed on each node, and to return some information based on the result of the operation. We have implemented the operations that compare some resource with a threshold value. For example, `filter load G 20.0` will return the list of nodes that have a CPU load greater than 20.0. To demonstrate the utility of the command, we have also implemented a more powerful *grep* operation. For example, `filter grep <keyword> <file>` will return the list of nodes on which the keyword `<keyword>` has occurred in the file `<file>`. This *grep* operation can be used for diagnosing failures of distributed applications. In fact, we have frequently used it to see if our MON server has reported some error message on some nodes.

Software Push When running an application on a large distributed system, a user may need to push the application code to a large number of nodes from time to time. Our on-demand overlays can also be used for such software push [2]. A tree structure is unsuited for software push, because the downloading rate of a node is limited by that of its parent. Therefore, we use DAG structures for software push, so that each node can download data from multiple parents at the same time. The DAG structure also improves the failure resilience of the system, because the downloading of a node is not affected by a parent failure, as long as it has other parents.

When a node can download data from multiple parents at the same time, some kind of coordination is needed between the parents. In our MON design, we adopt a *multi-parent, receiver driven* downloading approach similar to those used in recent P2P streaming systems [19]. Figure 2 illustrates how the approach works. Suppose a node p has three parents, q_1, q_2 and q_3. The file to be downloaded is divided into blocks. Whenever a node downloads a block, it will notify its children about the new

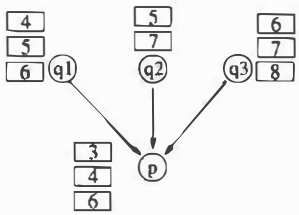


Figure 2: Multi-parent, receiver driven download

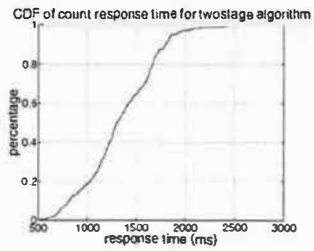


Figure 3: CDF of `count` time for twostage.

Table 1: Tree construction performance

	rand5	rand6	rand8	twostage
coverage	314.89	318.64	320.52	321.59
create time(ms)	3027.21	3035.46	2972.46	2792.03
count time(ms)	1539.19	1512.07	1369.92	1354.79

block. As a result, node p knows about the blocks that are available at each parent. Node p will then make a scheduling decision, and request different blocks to be received from different parents. For simplicity, we have used a "first fit" scheduling algorithm. For each parent, we request the first block that this parent can provide, and that is still needed by the requesting node. For example, for the scenario shown in Figure 2, node p will request block 5 from q_1, block 7 from q_2, and block 8 from q_3.

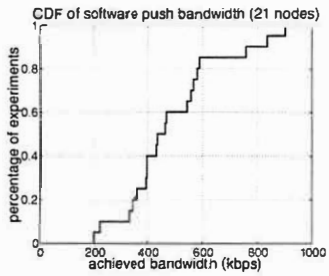
Our MON system uses UDP for most of the communication. However, for software push, we use TCP connections. Suppose a DAG has been created, a user can issue a `Push` command to push a file to all the nodes. Each MON server that receives a `Push` message from a parent will first establish a TCP connection to the parent, then create a TCP server socket to serve its children (if the server socket has not been created). Finally it will send the `Push` message to all of its children. Once all the children have established TCP connections to a node, it begins to request blocks from its parents, and advertise the downloaded blocks to its children.

3 Evaluation Results

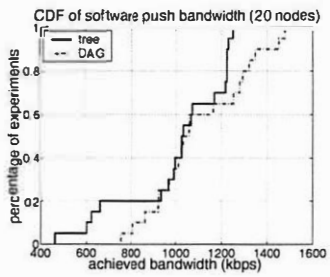
Our MON has been implemented and running on the PlanetLab for several months. The current deployment includes about 330 nodes. We have also created a web interface for people to try out MON [3]. In this section, we present some initial experiment results to evaluate our tree construction and software push algorithms.

Table 1 shows the performance of different tree construction algorithms. The experiments are conducted on our current MON deployment. For each algorithm, we create about 200 overlays and compute the average of the number of nodes covered (coverage), tree creation time, and the `count` response time. *rand5, rand6* and *rand8* are the random algorithm with $k = 5, 6$ and 8, respectively. *twostage* is the two stage algorithm with $k = 5$. The tables shows that the two stage algorithm has better performance compared with random tree construction. On average, the two stage algorithm can create a tree in less than 3 seconds and cover about 321 nodes [3]. And a simple `count` query takes about 1.3 seconds. In comparison, *rand5* covers only about 315 nodes, and its `count` response time is more than 180ms larger. Figure 3 shows the CDF of the `count` response time for the two stage algorithm. We can see that the response time is less than 1500ms about 65% of the time, and less than 2000ms about 97% of the time.

To evaluate our software push algorithm, we pick 21 PlanetLab nodes mostly from universities in North America, and push a file of 1MB to the nodes. The file is divided into 50KB blocks. We first push the file to each node directly from our local node, and measure the bandwidth achieved. The bandwidth between our local node and most of the nodes is between 1Mbps and 3Mbps. However, for one node (`planet2.cs.rochester.edu`) the bandwidth is less than 400Kbps [4]. We then create a DAG and push the file to all the nodes. We repeat the experiment for 20 times and show the CDF of the bandwidth in Figure 4(a). The figure shows that most of the time, we can achieve a bandwidth between 400Kbps and 600Kbps, and the average is about 490Kbps. Figure 4(b) shows the result when we remove `planet2.cs.rochester.edu`. We can see most of the time the bandwidth is between 900Kbps and 1.3Mbps, and the average is about 1.1Mbps. Since all the nodes are receiving the data at the same time, this means on average we can achieve an effective aggregate bandwidth of about 22Mbps, which is about 7 times the largest bandwidth that our local node can provide to an individual node (`planetlab2.cs.uiuc.edu`). The above experiments allowed each node to have a maximum of 3 parents (the actual number of parents may be smaller). Figure 4(b) also shows the result for 20 nodes when each node has at most 1 parent (i.e. trees). We can see that about 20% of the time, the bandwidth is less than 700Kbps, and on average the bandwidth is about 10% smaller than the DAG case. This shows the advantage of DAG based multi-parent downloading schemes.



(a) 21 nodes



(b) 20 nodes

Figure 4: Software push bandwidth of MON.

4 Discussions

Many useful tools have been developed to make a distributed system such as the PlanetLab easier to use. CoMon [1], Ganglia [15] and many other tools provide resource monitoring for each PlanetLab node. The Application Manager [4] can monitor the status of individual applications. SWORD [16] provides resource discovery services. And PIER [10, 11] allows SQL like queries in large scale networks. However, these systems generally do not allow a user to execute *instant management commands* pertaining their own applications. In contrast, the `filter` command of MON can potentially be used to execute any operations (e.g., shell commands) on a node, just like PSSH and vxargs. Different from these two tools, however, MON is based on a distributed overlay structure, thus it has better scalability and allows in-network aggregations [5].

Since MON makes it easier to execute simultaneous commands on large number of nodes, it is important to have built-in security mechanisms to prevent misuse of the system. Although MON currently does not have any authentication mechanism, it is relatively easy to use public key of the user for authentication, which is already available on the PlanetLab nodes. For example, each time an overlay is created on demand, the private key of the user is used to "encrypt" some information about the user, such as the slice name and IP address. A MON server will continue with the overlay construction only if it can verify the message using the slice's public key. Timestamps can be used to prevent replay of the message, and a session key can be included for the encryption and decryption of subsequent session messages. Although encryption/decryption may increase the end to end delay, we do not expect the impact to be significant.

5 Conclusion

We have presented the design and preliminary evaluation of MON, a management overlay network designed for large distributed applications. Different from existing tools, MON focuses on the ability of a user to execute *instant management commands* such as status query and software push, and builds *on-demand* overlay structures for such commands. The on-demand approach enables MON to be lightweight, failure resilient, and yet simple. Our results further demonstrate that MON has good performance, both in command response time and aggregate bandwidth for software push.

MON is an on going project and we are continuing working on it. Specifically, our software push component is not mature yet. We will improve the system and experiment on significantly larger scales. We will also explore other on-demand overlay construction algorithms, for example, those that cover a specified subset of nodes, and those that can scale to even larger networks.

References

[1] CoMon. http://comon.cs.princeton.edu/.

[2] CoTop. http://codeen.cs.princeton.edu/cotop/.

[3] MON. http://cairo.cs.uiuc.edu/mon/.

[4] Planetlab application manager. http://appmanager.berkeley.intel-research.net/.

[5] PSSH. http://www.theether.org/pssh/.

[6] Stork. http://www.cs.arizona.edu/stork/.

[7] vxargs. http://dharma.cis.upenn.edu/planetlab/vxargs/.

[8] R. Adams. Distributed system management: Planetlab incidents and management tools. PlanetLab Design Notes PDN-03-015.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *SOSP'03*, 2003.

[10] B. Chun, J. Hellerstein, R. Huebsch, P. Maniatis, and T. Roscoe. Design considerations for information planes. In *WORLDS'04*, December 2004.

[11] R. Huebsch, J. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, 2003.

[12] A.-M. Kermarrec, L. Massoulie, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transaction on Parallel and Distributed Systems*, 14(2), February 2003.

[13] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP'03*, October 2003.

[14] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. Demers. Decentralized schemes for size estimation in large and dynamic groups. In *IEEE Symp. Network Computing and Applications*, 2005.

[15] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30, July 2004.

[16] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on planetlab with sword. In *WORLDS'04*, December 2004.

[17] K. Park and V. S. Pai. Deploying large file transfer on an http content distribution network. In *WORLDS'04*, December 2004.

[18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *HotNets-I*, 2002.

[19] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. DONet: A data-driven overlay network for efficient live media streaming. In *IEEE INFOCOM'05*, Miami, FL, 2005.

Notes

[1] In practice, due to transient and permanent node failures, a user is often prepared if not all the desired nodes can be accessed.

[2] We note that previous research work including SplitStream [9], Bullet [13] and CoBlitz [17] has addressed the problem of content distribution from one node to a large number of receivers. Our goal in developing the software push component, however, is to provide a easy-to-use system that can be integrated with our status query component, so that the user can accomplish most management tasks with the MON deployment.

[3] Note although we deploy MON on about 330 nodes, the precise number of live nodes may vary during the experiment.

[4] Note the bandwidth is the "end-to-end" bandwidth that includes the initial delay for sending the `Push` message, creating TCP server sockets, and waiting for the child connections. The effect of this initial delay will become less significant for large files. For example, the bandwidth between our local node and `planet-lab2.cs.uiuc.edu` is about 3Mbps for 1MB files, and about 13Mbps for 8MB files.

[5] Aggregating the results of arbitrary operations may need some workaround. For example, instead of executing an operation on every node and return the result, we can execute the command, return the nodes on which the operation succeeded (or failed). This way only small amount of data is returned, which may be less distracting and more interesting to the user.

Supporting Network Coordinates on PlanetLab

Peter Pietzuch, Jonathan Ledlie, Margo Seltzer
Harvard University, Cambridge, MA
hourglass@eecs.harvard.edu

Abstract

Large-scale distributed applications need latency information to make network-aware routing decisions. Collecting these measurements, however, can impose a high burden. *Network coordinates* are a scalable and efficient way to supply nodes with up-to-date latency estimates. We present our experience of maintaining network coordinates on PlanetLab. We present two different APIs for accessing coordinates: a per-application library, which takes advantage of application-level traffic, and a stand-alone service, which is shared across applications. Our results show that statistical filtering of latency samples improves accuracy and stability and that a small number of neighbors is sufficient when updating coordinates.

1 Introduction

Collecting up-to-date latency measurements between nodes in an overlay network is important for many classes of applications. Proximity-aware distributed hash tables use latency measurements to reduce the delay stretch of lookups [15], content distribution systems construct network-aware trees to minimize dissemination times [1], and decentralized web caches need latency information to map clients to cache locations. Especially in a wide-area network, communication latencies have a significant impact on the overall execution time of operations.

To exploit network locality, today's overlay networks are left with the burden of performing their own network measurements. Developers must continually reinvent the wheel duplicating measurements when multiple network-aware overlays are sharing a single distributed testbed, such as PlanetLab [19]. Implementations that gather all-pairs latency measurements are only scalable for relatively small overlay deployments. For example, the all-pairs ping service managed by Stribling [18] has recently ceased operation because it became infeasible to obtain up-to-date measurements for over 500 PlanetLab nodes. In addition, measuring techniques that lead to good latency samples without suffering from high variance caused by measurement anomalies are non-trivial.

To address these issues, a *latency service* can provide applications with up-to-date estimates of network latencies between nodes. We describe our experience of maintaining such a service on PlanetLab based on *network coordinates*. Here, each overlay node maintains a coordinate obtained through an embedding of latency measurements in a metric space. The Euclidean distance between two coordinates is an estimate of the communication latency between the nodes. This enables nodes to infer latencies to remote nodes without the overhead of a direct latency measurement. The metric space interpolates non-existing measurements, which reduces the measurement overhead from $O(n^2)$ to linear in the number of nodes.

We discuss trade-offs between two different solutions for a network coordinate service: a dedicated, stand-alone *service*, which is shared among applications, and a per-application *library*, which exploits application-specific traffic for network coordinate updates. Our experience deploying network coordinates on PlanetLab reveals that coordinate stability and convergence is a challenge. We have developed two techniques to address this: statistical filtering of latency samples and decoupling low-level coordinate updates from the coordinates used by applications. We have found that our implementation of a latency service now provides network coordinates that are sufficiently stable and accurate to support our application needs, while keeping the measurement overhead small.

After a survey of existing work in Section 2, we present the APIs and trade-offs of our network coordinate service and library in Section 3. In Section 4, we show how statistical filtering and our delayed update technique greatly improves accuracy and stability and how a small number of measurement neighbors can lead to accurate coordinates. We conclude in Section 5.

2 Latency Service

A latency service enables overlay nodes to obtain latency estimates to other nodes. We adopt the following design goals for our latency service.

1. **Good accuracy.** Latency estimates between nodes should have a relatively low error but the required accuracy depends on the application. For example, if the latency estimate is used to select the nearest node, a certain error is tolerable as long as it does not affect the result. The latency service must also achieve its accuracy goal when network latencies are changing due to BGP route updates or congestion.
2. **Low measurement overhead.** The latency service should minimize latency probing to conserve network resources. Latency measurements should use application data packets between nodes when possible. Note that there is a tension between the achievable accuracy and the measurement overhead.
3. **Quick latency prediction.** Many applications require quick decisions based on latencies between nodes. The latency service itself should not introduce a long delay when queried for latency estimates.
4. **Scalability.** The design of the latency service must be

scalable in terms of the number of nodes in the network for which latency measurement are required.

5. **Simple application integration.** It should be easy to run the latency service and for an application node to obtain latency estimates. The latency service should have an intuitive API and any node should be able to use the latency service.

2.1 Previous Work

Several research groups have recognized the need for a latency service on the Internet. Unfortunately, many current proposals for latency services make a poor trade-off between accuracy and overhead, are not widely deployed, require changes to the network, or have scalability issues. In this section, we provide a survey of latency services and determine their compliance with our design goals.

The simplest latency service gathers all pairs latency information and makes this data available to all nodes via a centralized location, as exemplified by the *all pairs ping service* [18] on PlanetLab. Such an approach causes a large amount of measurement traffic because every node measures latencies to every other node.

IDMaps [5] is a latency service that attempts to minimize measurement traffic. It uses a network of *tracers* that proactively measure distances between themselves and to representative nodes from each address prefix. This information is used to create a virtual distance map of the Internet. Since only tracers measure latency, the overhead is kept low but the prediction error is determined by the distribution of tracer locations. Achieving a good distribution is hard because the physical network topology is not known in practice.

The *Internet Iso-bar* [2] system attempts to remove the requirement of topology knowledge by dividing network nodes into clusters depending on latencies. A node from each cluster is then selected to monitor intra- and inter-cluster latencies and to respond to latency queries. However, the accuracy of the system depends on how amenable the network is to clustering. The cluster size determines the measurement overhead.

Ratnasamy *et al.* propose a latency service that attempts to reduce the number of network measurements even further [14]. Nodes measure their network distance only to a small number of *landmark nodes* and use the results to partition themselves into *bins*. Nodes that fall within the same bin are deemed to be close. Although this scheme vastly reduces the measurement overhead compared to other systems, it also exhibits a high error due to the coarse-grained assignment to a fixed number of bins.

Nakao *et al.* observe that much of the network information that applications are interested in is already collected by lower network layers. They propose to exploit this through a *routing underlay* [11], which provides a standardized interface for applications to inspect the state and structure of the network. Although an underlay would provide efficient access to network information already gathered by routers, it requires changes to routers.

2.2 Network Coordinates

A latency service can be constructed using a *network embedding* [8, 12] that embeds measured latencies between nodes in a low-dimensional geometric space. Each node maintains a *network coordinate (NC)*, with the goal that the Euclidean distance between two NCs is an estimate of physical network latency. Two classes of algorithms were proposed to compute NCs: *landmark-based* schemes, such as *GNP* [12], *Lighthouses* [13], and *PIC* [3], which use a fixed number of landmark nodes for NC calculation, and *simulation-based* approaches, such as *Vivaldi* [4] and *BBS* [16], which model NCs as entities in a physical system. Since one of our design goals for the latency service is scalability, we adopt a fully-decentralized, simulation-based approach for our NC service.

VIVALDI($\vec{x_j}, w_j, l_{ij}$)

1 $w_s = \frac{w_i}{w_i + w_j}$

2 $\epsilon = \frac{|\|\vec{x_i} - \vec{x_j}\| - l_{ij}|}{l_{ij}}$

3 $\alpha = c_e \times w_s$

4 $w_i = (\alpha \times \epsilon) + ((1 - \alpha) \times w_i)$

5 $\delta = c_c \times w_s$

6 $\vec{x_i} = \vec{x_i} + \delta \times (\|\vec{x_i} - \vec{x_j}\| - l_{ij}) \times u(\vec{x_i} - \vec{x_j})$

Figure 1: *Vivaldi* update algorithm.

The *Vivaldi* algorithm calculates NCs as the solution to a spring relaxation problem. The measured latencies between nodes are modeled as the extensions of springs between massless bodies. A network embedding with a minimum error is found as the low-energy state of the spring system. Each node successively refines its NC through periodic updates with other nodes in its *neighbor set*.

Figure 1 shows how a new observation, consisting of the remote node's NC $\vec{x_j}$, its confidence w_j, and a latency measurement l_{ij} between the two nodes, i and j, is used to update the local NC. The *confidence* w_i quantifies how accurate a NC is believed to be. First, the *sample confidence* w_s (Line 1) and the *relative error* ϵ (Line 2) are calculated. The relative error ϵ expresses the accuracy of the NC in comparison to the true network latency. Second, node i updates its confidence w_i with an exponentially-weighted moving average (Line 4). The weight α is set according to the sample confidence w_s (Line 3). Also based on the sample confidence, δ dampens the change applied to the NC (Line 5). As a final step, the NC is updated in Line 6 (u is the unit vector). Constants c_e=c_c=0.25 affect the maximum impact an observation can have on confidence and NC, respectively [6]. We define the *stability* of a NC as its total change over time in ms/s.

3 Architecture

A latency service based on NCs exploits several properties of NCs that help satisfy the design goals from Section 2.

- NCs achieve good accuracy on Internet topologies. Although an embedding error arises because Internet latencies violates the triangle inequality, these violations are not severe enough to prevent a metric embedding in practice. Previous work [4] has found a median relative error of 11%, which we confirmed on PlanetLab.
- Non-existent measurements between nodes are interpolated by the network embedding, thus reducing the measurement overhead. The trade-off between measurement overhead and accuracy is made explicit by NCs. The accuracy and convergence of NCs can be improved by increasing the measurement frequency and

extending the neighbor set.

- NCs provide almost instantaneous latency predication because they do not actively initiate new latency measurements to respond to latency queries. Active measurement approaches, such as Meridian [20], may introduce a non-trivial delay while a fresh latency estimate is being obtained.
- The decentralized algorithm for computing NCs makes the implementation scalable to a large number of nodes. We have successfully deployed a NC service on over 300 PlanetLab nodes.

To achieve simple application integration, we propose two different architectures: a stand-alone *NC service* and a per-application *NC library*. Both approaches have the advantage that they provide a correct implementation of NC to applications. As will be explained in Section 4, the application programmer does not have to deal with the complexity of latency measurement.

Network Coordinate Service. If the network infrastructure is cooperative and under control of a single authority, such as PlanetLab, an efficient solution is to deploy a *NC service* on all the nodes. Each application then accesses the locally running NC service. This has the advantage that the cost of inter-node measurements is amortized across all applications that share the service. A drawback of this approach is that parameters, such as the measurement frequency, which determines the convergence of the NCs, must be set globally for all applications.

```
double    estimateLat    (double[] remoteNC)
double[]  getNC          ()
double    getConfidence  ()
double    getRelError    ()
double    forceUpdate    (IPAddr remoteNode)
```

Above we show the API of the latency service that is part of our SBON deployment [17] on PlanetLab. The function `estimateLat` returns the latency estimate between a local and remote node given the remote node's NC. The local NC and confidence are returned by the `getNC` and `getConfidence` calls, respectively. A call to `getRelError` returns the current median relative error over the last n latency measurement that were used for coordinate updates. If the application needs an up-to-date latency to a remote node, a call to `forceUpdate` causes the NC service to perform a measurement to the remote node returning the observed latency. This API assumes that nodes in a distributed application are identified as an IP address and NC pair, `(IPAddr, NC)`. As a result, any node can obtain a latency estimate to another node about which it has learned.

Network Coordinate Library. In some cases, an application should include a module for latency estimation without relying on an externally running service. This is true for peer-to-peer applications that are deployed on a varying set of heterogeneous nodes. To address this, we also propose a *NC library* that any application can link against to support NCs. In order to avoid duplicating functionality, the library handles only the computation of coordinates but leaves the actual network communication for network probing to the application. This enables the application to exploit application traffic as much as possible for measurements.

```
void      updateNC       (IPAddr remoteNode,
                          double[] remoteNC,
                          double remoteConf,
                          double latency)
void      forceUpdate    (IPAddr remoteNode)
```

In addition to the functions provided by the stand-alone service, the NC library API has a function `updateNC` that is used by the application to feed in new network measurements from application-level traffic. Only if the application-level traffic is not frequent enough or does not cover a large enough set of nodes to compute an accurate NC does the library request additional latency measurements from the application. As will be explained in Section 4.2, the NC library monitors its relative error to decide if the NC is converging sufficiently. If this is not the case, it uses the `forceUpdate` callback to the application to request more diverse measurements by initiating a latency measurement to a new remote node.

4 Implementation Issues

Regardless of whether NCs are accessed through a service or a library, they must be designed to handle practical networking problems, such as measurement variation, data loss, and node failures. The focus of our work to date has been on measurement variation: creating an accurate and stable coordinate system using real world latency samples. In this section, we explain our solutions to handle non-ideal latency samples, which have a significant negative impact when left unfiltered. We also describe how measurement overhead can be controlled by tuning neighbor sets. As an overview, we found that:

- Latency samples for a particular link have a high variance. These raw samples can cause wild, temporary perturbations, which cascade across the coordinate space. We found that a statistical filter suppressed these anomalies and greatly improved accuracy and stability.
- Nodes' relative latencies change over time. We refine a filter to remove bad measurements while preserving changes in the underlying network. This sustains accuracy over time.
- Applications prefer stable coordinates. A distinction between *system-level NCs*, which are raw Vivaldi coordinates, and *application-level NCs*, which summarize a recent window of coordinate updates, help suppress unnecessary application activity. A new coordinate is only externally visible to an application after a significant change has occurred.

4.1 Measuring Latency

During our initial deployment of NCs on PlanetLab, we observed latency samples of as much as three orders-of-magnitude greater than the normal latency for a given link. When used for NC calculation, these samples induce a large coordinate change in a high confidence node, which, in turn, causes large shifts in the NC of its neighbors. Such changes keep propagating through the coordinate space, causing high instability, low convergence, and decreased accuracy, because coordinate shifts are not reflecting future measurements. Occasionally, but not always, we could attribute large values in our application-level measurements to high CPU load on one of the nodes.

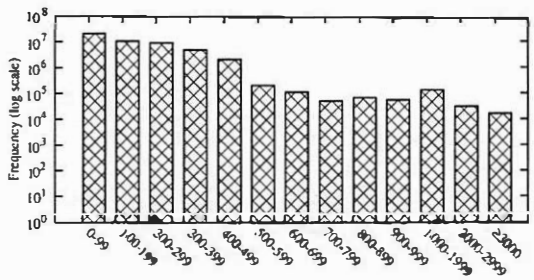


Figure 2: Raw latency samples on PlanetLab (ms).

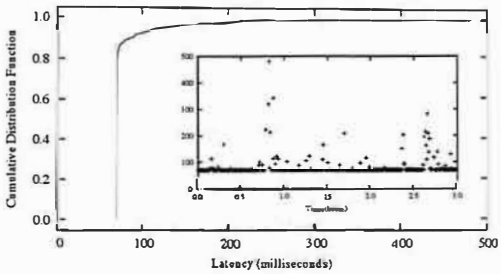


Figure 3: Kernel-level ping measurements.

To illustrate the extent of the anomalies, we show a distribution of application-level UDP latency samples between 269 nodes collected over three days on PlanetLab in Figure 2. Over 0.4% of the samples are greater than one second, larger than even a slow intercontinental link, and frequent enough to periodically distort the coordinate space. Not only is a significant fraction of samples large, but also individual links have extended tails: samples of a link tend to produce a consistent latency within a tight range, but then a tail of the samples can extend into the tens of seconds. Both the range and tail depend on the link. We found that feeding these raw samples directly into Vivaldi leads to poor accuracy and stability.

We also tried using kernel-level ping measurements and found that they suffered from a similar baseline and extended tail. Figure 3 shows the results of a three hour set of ping measurements using the `ping` program between two PlanetLab nodes (`berkeley` to `uvic.ca`). The data shows that 82% of the samples fall within 1ms of the median, but that the largest 5% are 2–7 times the median. Even though the measurements are being timestamped by the kernel, there are many large measurements that would jolt a stable coordinate system. In addition, as the subgraph shows, the deviations from the baseline measurement are not clustered all at one time, but occur throughout the trace; they do not signal shifts, but aberrations. Because kernel-level measurements would need to be filtered also and do not have the benefit of application-level traffic, we decided to find a way to incorporate samples with a high variance into the NC computation.

Although we estimate that approx. 90% of links fall into the type shown in Figure 3, a small percentage do exhibit multi-modal behavior. If multi-modal behavior was on a short time scale, it would be unclear what value would be appropriate to feed into the NC update algorithm; it might perhaps require a more complicated link description (*e.g.*, a PDF). However, we have not seen

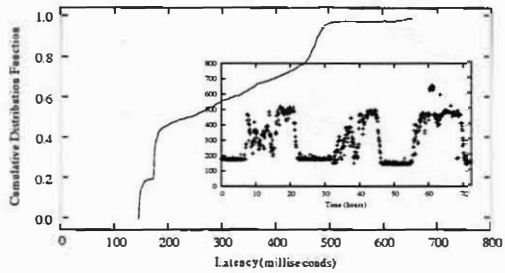


Figure 4: Long-term, periodic, bimodal latency samples.

this behavior in practice. Instead, we have found cases of long-term, periodic bi-modal link latencies, as shown in Figure 4 (`ntu.edu.tw` to `6planetlab.edu.cn`). That this behavior is long-term is important for two reasons: (1) it appears reasonable to summarize each link with a single current baseline latency and (2) NCs need continuous maintenance because this baseline latency changes over time.

Statistical Filtering. We explored three obvious but ineffective approaches before arriving at our final solution for latency signal extraction. First, we tried using *simple thresholds*: if an observation is larger than a constant, it is ignored. This did not work because one link's normal latency was well into the range of the tail of another link. Second, we applied an *exponentially-weighted moving average (EWMA)* to each link's sample history. We found that this performed worse than no filter at all because it weighted outliers too strongly, even with unusually low weights. Finally, we tried a more Vivaldi-specific solution: *lowering confidence* in response to high load. However, because sample variance can only partially be attributed to load, this solution was also not effective.

Instead, we found that a non-linear *moving percentile (MP)* filter greatly improved accuracy and stability. The MP filter takes two parameters: a window size of samples and the percentile of these samples to output. It removes noise and, based on the window size, responds to changes in the underlying signal. Before presenting our experimental results, we introduce a technique layered on top of the filtered raw coordinate.

Application-Level NCs. Our latency service makes a distinction between *system-level* and *application-level* NCs. The former are raw Vivaldi coordinates, which are updated with each observation. The latter are the application's idea of the local NC, updated only when a statistically significant change in the system-level NC has occurred. While some applications may want to access the raw value, many others prefer updates when the system-level NC exhibits sustained change compared to its past.

We found two successful heuristics for setting application-level NCs, both based on a change detection algorithm that uses sliding windows [7]: RELATIVE and ENERGY. Both compare a current window of coordinates to a window starting at the most recent application-level coordinate update. RELATIVE compares the two windows based on the amount of change relative to the nearest known neighbor. ENERGY compares them based on a statistical test that measures the Euclidean distance between two multidimensional distributions. Both update the application-level coordinate to the centroid of a re-

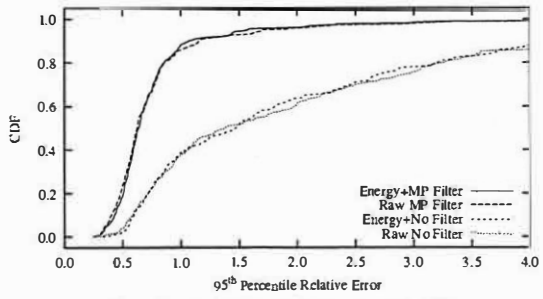


Figure 5: Accuracy on PlanetLab.

cent window of coordinates and both heuristics allow the raw coordinate to "float" in a given region. As long as the coordinate does not leave the region and other major changes in the network do not occur, application updates will be suppressed. Application-level NCs increase stability without decreasing accuracy.

Accuracy Results. Experimentally, we determined that a low percentile (*e.g.,* 25^{th}) and a window size of 4–8 samples or larger gives good results with the latency samples seen on PlanetLab. Links are moderately consistent: most follow the pattern seen in Figure 3, but about 10% that in Figure 4. Windows that are too large suppress network changes that should be reflected in the NCs: short windows are more effective than long ones, keeping the required state low. Figure 5 shows results from using the MP filter and the ENERGY heuristic with 270 PlanetLab nodes. It shows that with the MP filter only 14% of the nodes experience a 95^{th} percentile relative error greater than one, while 62% of those without the filter do. The enhancements combine to reduce the median of the 95^{th} percentile relative error by 54%.

In this experiment we measure accuracy as the coordinate's ability to predict the next sample along that link. For each observation, we compute the relative error, that is, the difference between the predicted and actual latency, divided by the actual latency. Each node then has a collection of relative errors from its samples; the figure shows the 95^{th} percentile out of this distribution, collected for the second half of a 4 hour run.

Defining accuracy as relative error produces a low-level metric that may not sufficiently capture application impact. Recently, Lua *et al.* proposed **relative rank loss** (*rrl*) to calculate how well coordinates capture the relative ordering of (all) pairs of neighbors [10]. Thus, for each node x, if $(d_{xi} > d_{xj} \wedge l_{xi} < l_{xj})$ or $(d_{xi} < d_{xj} \wedge l_{xi} > l_{xj})$, then the distances d between coordinates have to led to an incorrect prediction of the relative latencies l, presumably inducing an application-level penalty due to the wrong preference of a farther node. While *rrl* quantifies the *probability* of incorrect rankings, we wanted a metric that captures the *magnitude* of each rank misordering as well. For some applications, choosing the absolute nearest neighbor is important; however, often the extent of the error should be penalized: an error of 1ms is less severe than one of 100ms. **Weighted rrl** (*wrrl*) captures this by taking the sum of the latency penalties l_{ij} of pairs ranked incorrectly, normalized over all possible latency penalties. However, *wrrl* does not express the percentage in lost latency that an application will notice when

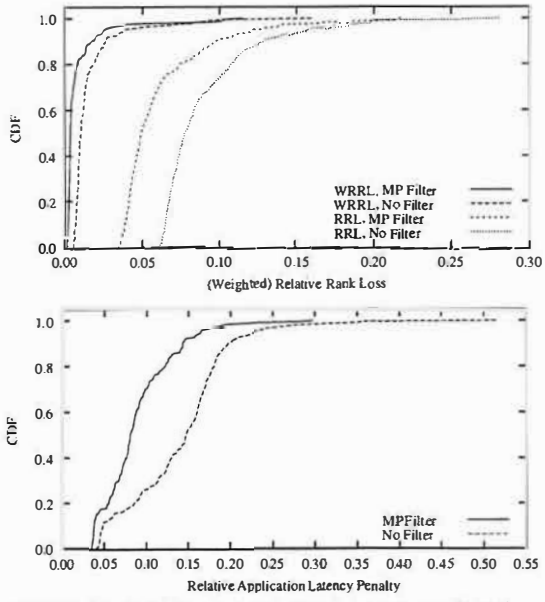


Figure 6: Application-oriented Accuracy Metrics.

using NCs. To approximate this quantity, we sum the relative latency penalty l_{ij}/l_{xi} for all pairs that are incorrectly ranked; we call this third metric the **relative application latency penalty** (*ralp*).

We illustrate how the MP filter affects these three metrics in Figure 6. The top graph portrays that while the probability of incorrect rankings (*rrl*) can range up to almost 30% for the worst case node, the latency penalty due to incorrectly ranked neighbors (*wrrl*) is 11% of the maximum in the worst case. The median *ralp* metric is 15% when using raw latency inputs, improving to 8% with the filter. We computed the "true" latency between nodes as the median for that link. In summary, our results indicate that the MP filter improves NC accuracy on PlanetLab for application-oriented operations, such as node ranking.

Stability Results. Unstable coordinates are problematic. Consider the situation where a node's coordinate is moving in a circle compared to using the centroid of that circle. If one is using the coordinate for a one-time decision (*e.g.,* finding the nearest node to initialize a Pastry routing table or finding a nearby web cache), unstable coordinates make a good decision less likely because it depends on the particular time the coordinates are compared. When coordinates are used for periodic decisions (*e.g.,* a proximity-based routing table update or re-positioning processing operators in a stream-based overlay), changing them may involve application-level work; unstable coordinates will induce updates based simply on their instability, not any fundamental change in relative node positions.

We measure stability as the amount of change in coordinates per unit time in *ms/sec*. This captures the amount of oscillation around a particular coordinate. Both the MP filter and application-level coordinates serve to suppress insignificant change. As shown in Figure 7, ENERGY dampens the filter's updates: 91% of the time it falls below even the minimum instability of the raw filter. Combined, the median instability is reduced by 96%. More detail on the MP filter and on the application-update heuristics can be found in our technical report [9].

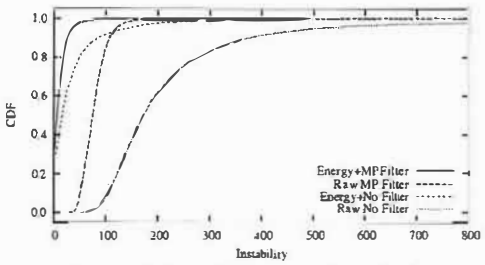


Figure 7: Instability on PlanetLab.

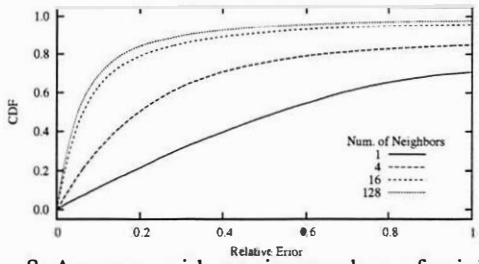


Figure 8: Accuracy with varying numbers of neighbors.

4.2 Limiting Measurement Overhead

One of the advantages of the NC library is that it takes advantage of application-level traffic to keep NCs up-to-date. This implies that a lack of samples must induce additional measurements to more nodes, but only when accuracy can be significantly improved. If nodes have a small neighbor set (*e.g.*, two), their accuracy to their neighbors and confidence w_i is high, but their accuracy to the rest of the system (overall accuracy) is low. As the number of neighbors increases, confidence and accuracy to neighbors decrease slightly, but overall accuracy improves.

In Figure 8, we show how overall accuracy varies with the number of neighbors. Accuracy increases asymptotically as the number of neighbors approaches the number of nodes. As shown, only 16 neighbors is a sufficiently good substitute for a fully connected graph. This means that regular application-level traffic to a small number of nodes is sufficient to support NCs on PlanetLab.

The NC library must decide when adding neighbors would significantly increase accuracy. However, a node cannot know its accuracy only by examining the relative error to its neighbors. Instead, it must estimate the overall relative error to all nodes. We propose that a node periodically samples a random node to test the current accuracy of its NC. If the tested accuracy is below a threshold, which is based on the expected accuracy of NCs on the Internet, it is likely that an increase of the neighbor set will reduce the relative error. The test node is then added permanently to the neighbor set. Similarly, if removing a node temporarily does not decrease accuracy (again by sampling a new node), the decrease is made permanent.

5 Conclusions

Up-to-date latency information as provided by a latency service is crucial for many distributed applications. Network coordinates are an efficient and scalable mechanism for obtaining latency estimates. However, any practical implementation must handle the variance of latency samples and minimize measurement overhead, while ensuring stable and accurate coordinates. In this paper, we have described the APIs of a network coordinate service and a library. We have also shown how statistical filtering addresses sample variance, how the distinction between system- and application-level coordinates improves coordinate stability, and how the use of application-level traffic for coordinate updates can reduce overhead. We believe that a network coordinate service can add network awareness to a wide range of applications and become one of a set of standard services for planetary-scale applications.

References

[1] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A Large-scale and Decentralized App-level Multicast Infrastructure. *JSAC*, 20(8), Oct. 2002.

[2] Y. Chen, K. H. Lim, R. H. Katz, and C. Overton. On the Stability of Network Distance Estimation. *SIGMETRICS Perform. Eval. Rev.*, 30(2), 2002.

[3] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical Internet Coordinates for Distance Estimation. In *Proc. of ICDCS'04*, Tokyo, Japan, Mar. 2004.

[4] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proc. of ACM SIGCOMM'04*, Portland, OR, Aug. 2004.

[5] P. Francis, S. Jamin, V. Paxson, et al. An Architecture for a Global Internet Host Distance Estimation Service. In *Proc. of INFOCOM'99*, New York, NY, Mar. 1999.

[6] T. Gil, F. Kaashoek, J. Li, et al. p2psim. `www.pdos.lcs.mit.edu/p2psim`.

[7] D. Kifer, S. Ben-David, and J. Gehrke. Detecting Change in Data Streams. In *Proc. of the 30th Int. Conf. on Very Large Data Bases*, Toronto, Canada, August 2004.

[8] J. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and Embedding Using Small Sets of Beacons. In *Proc. of FOCS'04*, Rome, Italy, Oct. 2004.

[9] J. Ledlie and M. Seltzer. Stable and Accurate Network Coordinates. TR 17-05, Harvard University, July 2005.

[10] E. K. Lua, T. Griffin, M. Pias, H. Zheng, and J. Crowcroft. On the Accuracy of Embeddings for Internet Coordinate Systems. In *Proc. of IMC'05*, Berkeley, CA, Oct. 2005.

[11] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. of the ACM SIGCOMM'03 Conference*, Karlsruhe, Germany, Aug. 2003.

[12] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. of INFOCOM'02*, New York, NY, June 2002.

[13] M. Pias, J. Crowcroft, S. Wilbur, S. Bhatti, and T. Harris. Lighthouses for Scalable Distributed Location. In *Proc. of IPTPS'03*, Berkeley, CA, Feb. 2003.

[14] S. Ratnasamy, P. Francis, M. Handley, B. Karp, and S. Shenker. Topology-Aware Overlay Construction and Server Selection. In *Proc. of INFOCOM'02*, June 2002.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of Middleware'01*, Nov. 2001.

[16] Y. Shavitt and T. Tankel. Big-Bang Simulation for Embedding Network Distances in Euclidean Space. In *Proc. of INFOCOM'03*, San Francisco, CA, Mar. 2003.

[17] Stream-Based Overlay Network. `www.eecs.harvard.edu/~prp/research/sbon`, Feb. 2005.

[18] J. Stribling. All-Pairs-Pings for PlanetLab. `www.pdos.lcs.mit.edu/~strib/pl_app`, Sept. 2004.

[19] The PlanetLab Consortium. PlanetLab. `www.planet-lab.org`, 2002.

[20] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *Proc. of SIGCOMM'05*, Aug. 2005.

This material is based upon work supported by the National Science Foundation under Grant No. 0330244.

Fixing the Embarrassing Slowness of OpenDHT on PlanetLab

Sean Rhea, Byung-Gon Chun, John Kubiatowicz, and Scott Shenker
University of California, Berkeley
opendht@opendht.org

1 Introduction

The distributed hash table, or DHT, is a distributed system that provides a traditional hash table's simple put/get interface using a peer-to-peer overlay network. To echo the prevailing hype, DHTs deliver incremental scalability in the number of nodes, extremely high availability of data, low latency, and high throughput.

Over the past 16 months, we have run a public DHT service called OpenDHT [14] on PlanetLab [2], allowing any networked host to perform puts and gets over an RPC interface. We built OpenDHT on Bamboo [13] and shamelessly adopted other techniques from the literature—including recursive routing, proximity neighbor selection, and server selection—in attempt to deliver good performance. Still, our most persistent complaint from actual and potential users remained, "It's just not fast enough!"

Specifically, while the long-term median latency of gets in OpenDHT was just under 200 ms—matching the best performance reported for DHASH [5] on PlanetLab---the 99th percentile was measured in seconds, and even the median rose above half a second for short periods.

Unsurprisingly, the long tail of this distribution was caused by a few arbitrarily slow nodes. We have observed disk reads that take tens of seconds, computations that take hundreds of times longer to perform at some times than others, and internode ping times well over a second. We were thus tempted to blame our performance woes on PlanetLab (a popular pastime in distributed systems these days), but this excuse was problematic for two reasons.

First, peer-to-peer systems are supposed to capitalize on existing resources not necessarily dedicated to the system, and do so without extensive management by trained operators. In contrast to managed, cluster-based services supported by extensive advertising revenue, peer-to-peer systems were supposed to bring power to the people, even those with flaky machines.

Second, it is not clear that the problem of slow nodes is limited to PlanetLab. For example, the best DHASH performance on the RON testbed, which is smaller and less loaded than PlanetLab, still shows a 99th percentile get latency of over a second [5]. Furthermore, it is well known that even in a managed cluster the distribution of individual machines' performance is long-tailed. The performance of Google's MapReduce system, for example, was improved by 31% when it was modified to account for a few slow machines its designers called "stragglers" [6]. While PlanetLab's performance is clearly worsened by the fact that it is heavily shared, the current trend towards utility computing indicates that such sharing may be common in future service infrastructures.

It also seems unlikely that one could "cherry pick" a set of well-performing hosts for OpenDHT. The MapReduce designers, for example, found that a machine could suddenly become a straggler for a number of reasons, including cluster scheduling conflicts, a partially failed hard disk, or a botched automatic software upgrade. Also, as we show in Section 2, the set of slow nodes isn't constant on PlanetLab or RON. For example, while the 90% of the time it takes under 10 ms to read a random 1 kB disk block on PlanetLab, over a period only 50 hours, 235 of 259 hosts will take over 500 ms to do so at least once. While one can find a set of fast nodes for a short experiment, it is nearly impossible to find such a set on which to host a long-running service.

We thus adopt the position that the best solution to the problem of slow nodes is to modify our algorithms to account for them automatically. Using a combination of delay-aware routing and a moderate amount of redundancy, our best technique reduces the median latency of get operations to 51 ms and the 99th percentile to 387 ms, a tremendous improvement over our original algorithm.

In the next section we quantify the problem of slow nodes on both PlanetLab and RON. Then, in Sections 3 and 4, we describe several algorithms for mitigating the effects of slow nodes on end-to-end get latency and show their effectiveness in an OpenDHT deployment of approximately 300 PlanetLab nodes. We conclude in Section 5.

2 The Problem of Slow Nodes

In this section, we study the problem of slow nodes in PlanetLab as compared to a cluster of machines in our lab. Our PlanetLab experiments ran on all the nodes we were able to log into at any given time, using a slice dedicated to the experiment.

Our cluster consists of 38 IBM xSeries 330 1U rackmount PCs, each with two 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM, and two 36 GB IBM UltraStar 36LZX hard drives. The machines use a single Intel PRO/1000 XF gigabit Ethernet adaptor to connect to a Packet Engines PowerRail gigabit switch. The operating system on each node is Debian GNU/Linux 3.0 (woody),

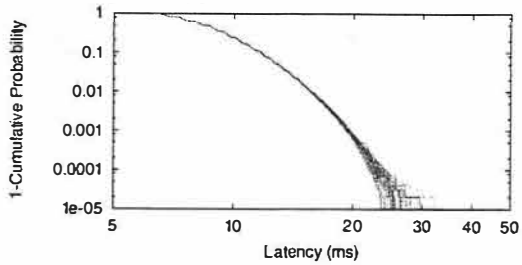


Figure 1: Time to compute a 128-bit RSA key pair on our cluster.

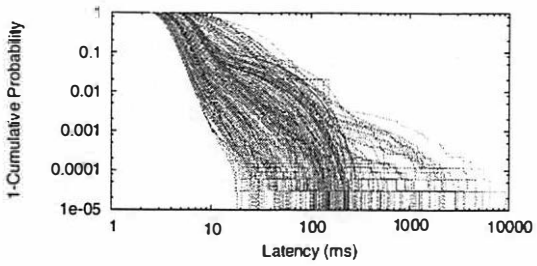


Figure 2: Time to compute a 128-bit RSA key pair on PlanetLab.

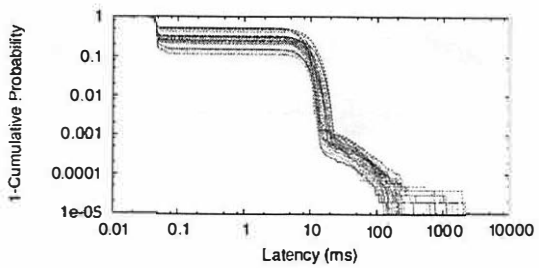


Figure 3: Time to read a random 1 kB disk block on our cluster.

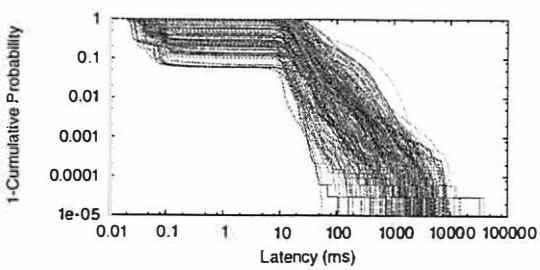


Figure 4: Time to read a random 1 kB disk block on PlanetLab.

running the Linux 2.4.18 SMP kernel. The two disks run in software RAID 0 (striping) using md raidtools-0.90. During our experiments the cluster is otherwise unused.

We study slowness with respect to computation time, network round-trip latency, and disk read latency. For each test, we wrote a simple C program to measure the latency of the resource under test. Our computation benchmark measures the latency to compute a new 128-bit RSA key pair, our network benchmark measures the round-trip time (RTT) of sending a 1 kB request and receiving a 1 kB response, and our disk benchmark measures the latency of reading a random 1 kB block out of a 1 GB file.

Figures 1--4 present the results of the computation and disk read tests; each line in these figures represents over 100,000 data points taken on a single machine. Figures 5 and 6 show the results of the network test; here each line represents over 10,000 data points taken between a single pair of machines.

Looking first at the cluster results, we note that operations we expect to be quick are occasionally quite slow. For example, the maximum ping time is 4.8 ms and the maximum disk read time is 2.5 seconds, a factor of 26 or 54,300 larger than the median time in each case.

Furthermore, there is significant variance between machines or pairs of machines (in the case of network RTTs). For example, the fraction of disk reads served in under 1 ms (presumably out of the cache) varies between 47% and 89% across machines. Also, one pair of machines never sees an RTT longer than 0.28 ms, while another pair sees a maximum RTT of 4.8 ms.

Based on these data, we expect that even an isolated cluster will benefit from algorithms that take into account performance variations between machines and the time-varying performance of individual machines. The MapReduce experience seems to confirm this expectation.

Turning to the PlanetLab numbers, the main difference is that the scheduling latencies inherent in a shared testbed increase the unpredictability of individual machines' performance by several orders of magnitude. This trend is most evident in the computation latencies. On the cluster, most machines showed the same, reasonably tight distribution of computation times; on PlanetLab, in contrast, a computation that never takes more than 18 ms on one machine takes as long as 9.3 *seconds* on another.

Unfortunately, very few nodes in PlanetLab are always fast, as shown in Figure 7. To produce this figure, we ran the disk read test on 259 PlanetLab nodes for 50 hours, pausing five seconds between reads. The figure shows the number of nodes that took over 100 ms, over 500 ms, over 1 s, or over 10 s to read a block since the start of measurement. In only 6 hours, 184 nodes take over 500 ms at least once; in 50 hours, 235 do so.

Furthermore, this property does not seem to be unique to PlanetLab. Figure 8 shows a similar graph produced from a trace of round-trip times between 15 nodes on RON [1], another shared testbed. We compute for each node the median RTT to each of the other fourteen, and rank nodes by these values. The lower lines show the values for the eighth largest and second largest values over time, and the upper line shows the size of the set of nodes that have ever had the largest or second largest value. In only 90 hours, 10 of 15 nodes have been in this set. This graph shows that while the aggregate performance of the 15 nodes is relatively stable, the ordering (in terms of per-

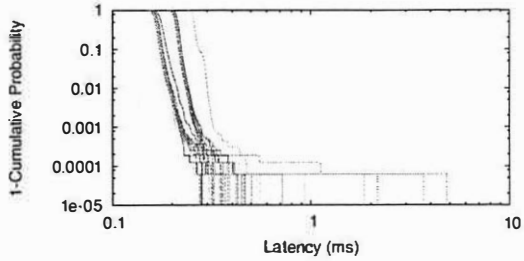


Figure 5: Time to send and receive a 1 kB message on our cluster.

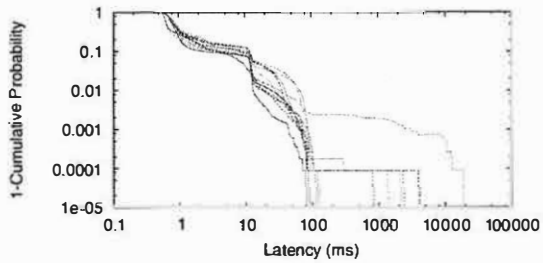


Figure 6: Time to send and receive a 1 kB message on PlanetLab.

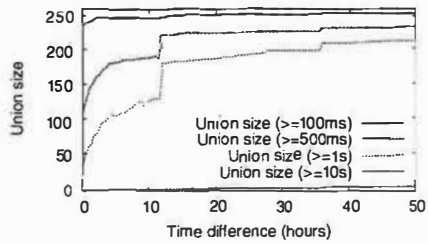


Figure 7: Slow disk reads on PlanetLab over time.

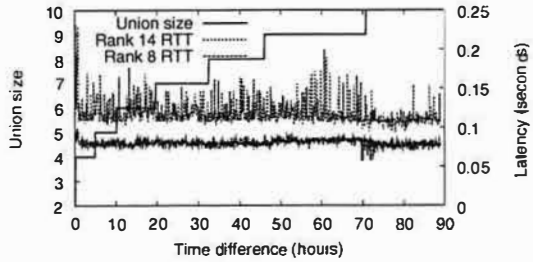


Figure 8: Slow round-trip times on RON over time.

formance) among them changes greatly.

In summary, with respect to network RTT and disk reads, both the relative performance of individual machines and the performance of a single machine over time can show significant variation even on an isolated cluster. On a shared testbed like PlanetLab or RON, this variation is even more pronounced, and the performance of computational tasks shows significant variation as well.

3 Algorithmic Solutions

Before presenting the techniques we have used to improve get latency in OpenDHT, we give a brief overview of how gets were performed before.

3.1 The Basic Algorithm

The key space in Bamboo is the integers modulo 2^{160}. Each node in the system is assigned an identifier from this space uniformly at random. For fault-tolerance and availability, each key-value pair (k, v) is stored on the four nodes that immediately precede and follow k; we call these eight nodes the *replica set* for k, denoted $R(k)$. The node numerically closest to k is called its *root*.

Each node in the system knows the eight nodes that immediately precede and follow it in the key space. Also, for each (base 2) prefix of a node's identifier, it has one neighbor that shares that prefix but differs in the next bit. This latter group is chosen for network proximity; of those nodes that differ from it in the first bit, for example, a node chooses the closest from roughly half the network.

Messages between OpenDHT nodes are sent over UDP and individually acknowledged by their recipients. A congestion-control layer provides TCP-friendliness and retries dropped messages, which are detected by a failure to receive an acknowledgment within an expected time. This layer also exports to higher layers an exponentially weighted average round-trip time to each neighbor.

To put a key-value pair (k, v), a client sends a put RPC to an OpenDHT node of its choice; we call this node the *gateway* for this request. The gateway then routes a put message greedily through the network until it reaches the root for k, which forwards it to the rest of $R(k)$. When six members of this set have acknowledged it, the root sends an acknowledgment back to the gateway, and the RPC completes. Waiting for only 6 of 8 acknowledgments prevents a put from being delayed by one or two slow nodes in the replica set. These delays, churn, and Internet routing inconsistencies may all cause some replicas in the set to have values that others do not. To reconcile these differences, the nodes in each replica set periodically synchronize with each other [12].

As shown in Figure 9, to perform a get for key k, the gateway G routes a get request message greedily through the key space until it reaches some node $R \in R(k)$. R replies with any values it has with key k, the set $R(k)$, and the set of nodes $S(k)$ with which it has synchronized on k recently. G pretends it has received responses from R and the nodes in $S(k)$; if these total five or more, it sends a response to the client. Otherwise, it sends the request directly to the remaining nodes in $R(k)$ one at a time until it has at least five responses (direct or assumed due to synchronization). Finally, G compiles a combined response

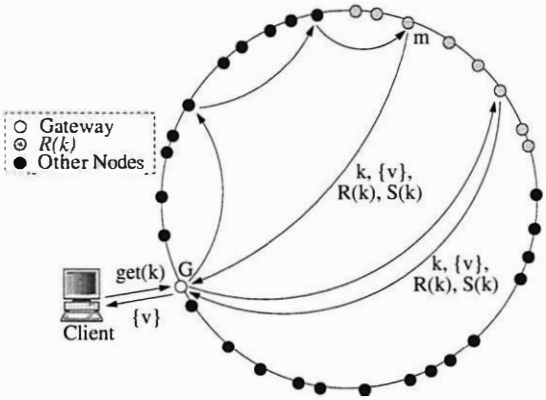


Figure 9: *A basic get request.*

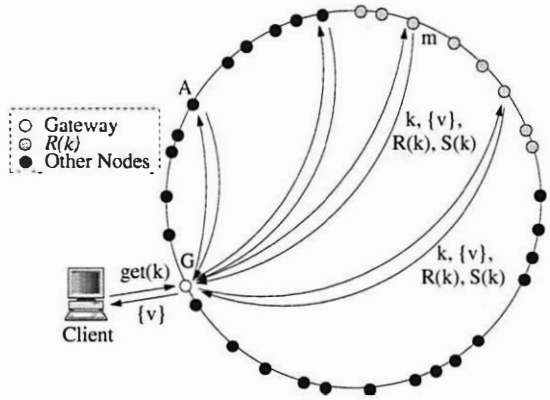


Figure 10: *An iterative get request.*

and returns it to the client.

By combining responses from at least five replicas, we ensure that even after the failure of two nodes, there is at least one node in common between the nodes that receive a put and those whose responses are used for a get.

3.2 Enhancements

We have explored three techniques to improve the latency of gets: delay-aware routing, parallelization of lookups, and the use of multiple gateways for each get.

3.2.1 Delay-Aware Routing

In the basic algorithm, we route greedily through the key space. Because each node selects its neighbors according to their response times to application-level pings, most hops are to nearby, responsive nodes. Nonetheless, a burst in load may render a once-responsive neighbor suddenly slow. Bamboo's neighbor maintenance algorithms are designed for stability of the network, and so adapt to such changes gradually. The round-trip times exported by the congestion-control layer are updated after each message acknowledgment, however, and we can use them to select among neighbors more adaptively.

The literature contains several variations on using such delay-aware routing to improve get latency. Gummadi et al. demonstrated that routing along the lowest-latency hop that makes progress in the key space can reduce end-to-end latency, although their results were based on simulations where the per-hop processing cost was ignored [7]. DHASH, in contrast, uses a hybrid algorithm, choosing each hop to minimize the expected overall latency of a get, using the expected latency to a neighbor and the expected number of hops remaining in the query to scale the progress each neighbor makes in the key space [5].

We have explored several variations on this theme. For each neighbor n, we compute ℓ_n, the expected round-trip time to the neighbor, and d_n, the progress made in the key space by hopping to n, and we modified OpenDHT to choose the neighbor n with maximum $h(\ell_n, d_n)$ at each hop, where h is as follows:

Purely greedy:	$h(\ell_n, d_n) = d_n$
Purely delay-based:	$h(\ell_n, d_n) = 1/\ell_n$
Linearly scaled:	$h(\ell_n, d_n) = d_n/\ell_n$
Nonlinearly scaled:	$h(\ell_n, d_n) = d_n/f(\ell_n)$

where $f(\ell_n) = 1 + e^{(\ell_n - 100)/17.232}$. This function makes a smooth transition for ℓ_n around 100 ms, the approximate median round-trip time in the network. For round-trip times below 100 ms, the nonlinear mode thus routes greedily through the key space, and above this value it routes to minimize the per-hop delay.

3.2.2 Iterative Routing

Our basic algorithm performs get requests *recursively*; routing each request through the network to the appropriate replica set. In contrast, gets can also be performed *iteratively*, where the gateway contacts each node along the route path directly, as shown in Figure 10. While iterative requests involve more one-way network messages than recursive ones, they remain attractive because they are easy to parallelize. As first proposed in Kademlia [9], a gateway can maintain several outstanding RPCs concurrently, reducing the harm done by a single slow peer.

To perform a get on key k iteratively, the gateway node maintains up to p outstanding requests at any time, and all requests are timed out after five seconds. Each request contains k and the Vivaldi [4] network coordinates of the gateway. When a node $m \notin R(k)$ receives a get request, it uses Vivaldi to compute ℓ_n relative to the gateway for each of its neighbors n, and returns the three with the largest values of $h(d_n, \ell_n)$ to the gateway.

When a node $m \in R(k)$ receives a get request, it returns the same response as in recursive gets: the set of values stored under k and the sets $R(k)$ and $S(k)$. Once a gateway has received a response of this form, it proceeds as in recursive routing, collecting at least five responses before compiling a combined result to send to the client.

3.2.3 Multiple Gateways

Unlike iterative gets, recursive gets are not easy to parallelize. Also, in both iterative and recursive gets, the gateway itself is sometimes the slowest node involved in a request. For these reasons we have also experimented with issuing each get request simultaneously to multiple gateways. This technique adds parallelism to both types of get, although the paths of the get requests may overlap as they near the replica set. It also hides the effects of slow gateways.

4 Experimental Results

It is well known that as a shared testbed, PlanetLab cannot be used to gather exactly reproducible results. In fact, the performance of OpenDHT varies on a hourly basis.

Despite this limitation, we were able to perform a meaningful quantitative comparison between our various techniques as follows. We modified the OpenDHT implementation such that each of the modes can be selected on a per-get basis, and we ran a private deployment of OpenDHT on a separate PlanetLab slice from the public one. Using a client machine that was not a PlanetLab node (and hence does not suffer from the CPU and network delays shown in Figures 2 and 6), we put into OpenDHT five 20-byte values under each of 3,000 random keys, reputting them periodically so they would not expire. On this same client machine, we ran a script that picks a one of the 3,000 keys at random and performs one get for each possible mode in a random order. The script starts each get right after the previous one completes, or after a timeout of 120 seconds. After trying each mode, the script picks a new key, a new random ordering of the modes, and repeats. So that we could also measure the cost of each technique, we further modified the OpenDHT code to record the how many messages and bytes it sends on behalf of each type of get. We ran this script from July 29, 2005 until August 3, 2005, collecting 27,046 samples per mode to ensure that our results cover a significant range of conditions on PlanetLab.

Table 1 summarizes the results of our experiments.

The first row of the table shows that our original algorithm, which always routes all the way to the root, takes 186 ms on median and over 8 s at the 99th percentile.

Rows 2–5 show the performance of the basic recursive algorithm of Section 3.1, using only one gateway and each of the four routing modes described in Section 3.2.1. We note that while routing with respect to delay alone improves get latency some at the lower percentiles, the linear and nonlinear scaling modes greatly improve latency at the higher percentiles as well. The message counts show that routing only by delay takes the most hops, and with each hop comes the possibility of landing on a newly slow node; the scaled modes, in contrast, pay enough attention to delays to avoid the slowest nodes, but still make quick progress in the key space.

Row	Parameters GW	I/R	p	Mode	Latency (ms) Avg	50th	90th	99th	Cost per Get Msgs	Bytes
1	1	Orig. Alg.			434	186	490	81 13	not measured	
2	1	R	1	Greedy	282	149	407	4409	5.5	1833
3	1	R	1	Prox.	298	101	343	5192	8.7	2625
4	1	R	1	Linear	201	99	275	3219	6.8	2210
5	1	R	1	Nonlin.	185	104	263	1830	6.0	1987
6	1	I	3	Greedy	157	116	315	788	14.6	3834
7	1	I	3	Prox.	477	335	1016	2377	33.1	6971
8	1	I	3	Linear	210	175	422	802	18.8	4560
9	1	I	3	Nonlin.	230	175	455	1103	18.3	4458
10	1	R	1	Nonlin.	185	104	263	1830	6.0	1987
11	2	R	1	Nonlin.	174	99	267	1609	6.0	1987
12	1–2	R	1	Nonlin.	107	71	171	609	11.9	3973
13	1	I	3	Greedy	157	116	315	788	14.6	3834
14	2	I	3	Greedy	147	110	294	731	14.6	3834
15	1–2	I	3	Greedy	88	70	195	321	29.3	7668
16	1–2	I	1	Greedy	141	96	289	638	13.9	4194
17	1–2	I	2	Greedy	97	78	217	375	22.5	6181
18	1–3	R	1	Nonlin.	90	57	157	440	16.8	5332
19	1–4	R	1	Nonlin.	81	51	142	387	22.4	7110
20	1–2	I	2	Greedy	105	84	232	409	20.2	5352
21	1–2	I	3	Greedy	95	76	206	358	26.5	6674
22	1–3	I	2	Greedy	86	62	196	332	30.3	8028

Table 1: *Performance on PlanetLab. GW* is the gateway, 1-4 for planetlab(14|15|16|13).millennium.berkeley.edu. *I/R* is for iterative or recursive. The costs of the single gateway modes are estimated as half the costs of using both.

We note that the median latencies achieved by all modes other than greedy routing are lower than the median network RTT between OpenDHT nodes, which is approximately 137 ms. This seemingly surprising result is actually expected; with eight replicas per value, the DHT has the opportunity to find the closest of eight nodes on each get. Using the distribution of RTTs between nodes in OpenDHT, we computed that an optimal DHT that magically chose the closest replica and retrieved it in a single RTT would have a median get latency of 31 ms, a 90th percentile of 76 ms, and a 99th percentile of 130 ms.

Rows 6–9 show the same four modes, but using iterative routing with a parallelism factor, p, of 3. Note that the non-greedy modes are not as effective here as for recursive routing. We believe there are three reasons for this effect. First, the per-hop cost in iterative routing is higher than in recursive, as each hop involves a full round-trip, and on average the non-greedy modes take more hops for each get. Second, recursive routing uses fresh, direct measurements of each neighbor's latency, but the Vivaldi algorithm used in iterative routing cannot adapt as quickly to short bursts in latency due to load. Third, our Vivaldi implementation does not yet include the kinds of filtering used by Pietzuch, Ledlie, and Seltzer to produce more accurate coordinates on PlanetLab [10]; it is possible that their implementation would produce better coordinates with which to guide iterative routing decisions.

Despite their inability to capitalize on delay-awareness, the extra parallelism of iterative gets provides enough resilience to far outperform recursive ones at the 99th percentile. This speedup comes at the price of a factor of two in bandwidth used, however.

Rows 10–12 show the benefits of using two gateways with recursive gets. We note that while both gateways are equally slow individually, waiting for only the quickest of them to return for any particular get greatly reduces latency. In fact, for the same cost in bandwidth, they far outperform iterative gets at all percentiles.

Rows 13–15 show that using two gateways also improves the performance of iterative gets, reducing the 99th percentile to an amazing 321 ms, but this performance comes at a cost of roughly four times that of recursive gets with a single gateway.

Rows 16–17 show that we can reduce this cost by reducing the parallelism factor, p, while still using two gateways. Using $p = 1$ gives longer latencies than recursive gets with the same cost, but using $p = 2$ provides close to the performance of $p = 3$ at only three times the cost of recursive gets with a single gateway.

Since iterative gets with two gateways and $p = 3$ use more bandwidth than any of the recursive modes, we ran a second experiment using up to four gateways per get request. Rows 18–22 show the results. For the same cost, recursive gets are faster than iterative ones at both the median and 90th percentile, but slower at the 99th.

These differences make sense as follows. As the gateways are co-located, we expect the paths of recursive gets to converge to the same replica much of the time. In the common case, that replica is both fast and synchronized with its peers, and recursive gets are faster, as they have more accurate information than iterative gets about which neighbor is fastest at each hop. In contrast, iterative gets with $p > 1$ actively explore several replicas in parallel and are thus faster when one discovered replica is slow or when the first replica is not synchronized with its peers, necessitating that the gateway contact multiple replicas.

5 Conclusions

In this work we highlighted the problem of slow nodes in OpenDHT, and we demonstrated that their effect on overall system performance can be mitigated through a combination of delay-aware algorithms and a moderate amount of redundancy. Using only delay-awareness, we reduced the 99th percentile get latency from over 8 s to under 2 s. Using a factor of four more bandwidth, we can further reduce the 99th percentile to under 400 ms and cut the median by a factor of three.

Since performing this study, we have modified the public OpenDHT deployment to perform all gets using delay-aware routing with nonlinear scaling, and we have encouraged users of the system to use multiple gateways for latency-sensitive gets. The response from the OpenDHT user base has been very positive.

Looking beyond our specific results, we note that there has been a lot of collective hand-wringing recently about the value of PlanetLab as an experimental platform. The load is so high, it is said, that one can neither get high performance from an experimental service nor learn interesting systems lessons applicable elsewhere.

We have certainly cast some doubt on the first of these two claims. The latencies shown in Table 1 are low enough to enable many applications that were once thought to be outside the capabilities of a "vanilla" DHT. For example, Cox et al. [3] worried that Chord could not be used to replace DNS, and others argued that aggressive caching was required for DHTs to do so [11]. On the contrary, even our least expensive modes are as fast as DNS, which has a median latency of around 100 ms and a 90th percentile latency of around 500 ms [8].

As to the second claim, there is no doubt that PlanetLab is a trying environment on which to test distributed systems. That said, we suspect that the MapReduce designers might say the same about their managed cluster. Their work with stragglers certainly bears some resemblance to the problems we have dealt with. While the question is by no means settled, we suspect that PlanetLab may differ from their environment mainly by degree, forcing us to solve problems at a scale of 300 nodes that we would eventually have to solve at a scale of tens of thousands of nodes. If this suspicion is correct, perhaps PlanetLab's slowness is not a bug, but a feature.

References

[1] `http://nms.csail.mit.edu/ron/data/`.

[2] A. Bavier et al. Operating system support for planetary-scale network services. In *NSDI*, Mar. 2004.

[3] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *IPTPS*, 2002.

[4] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[5] F. Dabek et al. Designing a DHT for low latency and high throughput. In *NSDI*, 2004.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[7] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM*, Aug. 2003.

[8] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *SIGCOMM IMW*, 2001.

[9] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *IPTPS*, 2002.

[10] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on PlanetLab. In *WORLDS*, 2005.

[11] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *SIGCOMM*, 2004.

[12] S. Rhea. *OpenDHT: A public DHT service*. PhD thesis, U.C. Berkeley, Aug. 2005.

[13] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Tech. Conf.*, 2004.

[14] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, , and H. Yu. OpenDHT: A public DHT service and its uses. In *SIGCOMM*, 2005.

(Re)Design Considerations for Scalable Large-File Content Distribution

Brian Biskeborn, Michael Golightly,* KyoungSoo Park, and Vivek S. Pai
Department of Computer Science
Princeton University

Abstract

The CoBlitz system was designed to provide efficient large file transfer in a managed infrastructure environment. It uses a content distribution network (CDN) coupled with a swarm-style chunk distribution system to reduce the bandwidth required at origin servers. With 6 months of operation, we have been able to observe its behavior in typical usage, and glean information on how it could be redesigned to better suit its target audience.

At its heart, this paper describes what happens when a plausible conceptual design meets the harsh realities of life on the Internet. We describe our experiences improving CoBlitz's performance via a range of techniques, including measurement-based feedback, heuristic changes, and new algorithms. In the process, we triple CoBlitz's performance, and we reduce the load it places on origin servers by a factor of five. In addition to improving performance for CoBlitz's users, we believe that our experiences will also be beneficial to other researchers working on large-file transfer and content distribution networks.

1 Introduction

Content distribution networks (CDNs) use distributed sets of HTTP proxies to serve and cache popular web content. They increase perceived web browsing speed by caching content at the edges of the network (close to end users), but they also provide a high degree of reliability when asked to serve very popular pages (by spreading load across many proxies instead of concentrating it on a single origin server). Large files such as movie trailers and free software ISO images are another popular form of content on the Internet, and they can place great strains on servers and network connections. CoBlitz is a service which layers efficient large-file distribution capabilities on top of the PlanetLab-based CoDeeN [15] CDN.

In a CDN such as CoDeeN, which runs on shared hosts owned by many different companies and educational institutions, the network infrastructure is very heterogeneous. Sites display a wide range of Internet connectivity, with available bandwidths ranging from a few hundred Kbps to almost 100 Mbps. In addition to this nonuniformity, the Internet itself is a hostile environment: TCP can take care of packet loss, but occasional congestion on high-capacity paths can slow data transfers to a crawl. In a system like CoBlitz, where large file requests are spread over numerous Web proxies, a few very slow downloads can have a significant impact on the overall download speed.

In this work, we describe the optimizations made to CoBlitz to improve its performance in the nonuniform environment of PlanetLab. Our changes have produced a significant increase in throughput, thus providing benefits for both public users and researchers. Futhermore, we think the lessons learned from CoBlitz apply, at least in part, to any project which aims to distribute content quickly and efficiently.

2 Background

From a user's perspective, CoBlitz provides a simple mechanism for distributing large files, by simply prefacing their URLs with a CoBlitz-enabling host name and port number. From an internal design perspective, CoBlitz is virtually the same as the CoDeploy system [9], which was designed to provide file synchronization across PlanetLab. Both systems use the same infrastructure for transferring data, which is layered on top of the CoDeeN content distribution network [15].

To briefly summarize CoDeeN's organization, each node operates independently, from peer selection to forwarding logic. Nodes periodically exchange heartbeats, which carry local node health information. The timings of these heartbeats allows nodes to determine network health as well as node overload. Nodes independently select peers using these heartbeats. All nodes also act as caches, and use the Highest Random Weight (HRW) algorithm [13] to determine which peer should receive requests that cannot be satisfied from the node's local cache. If a forwarded request cannot be satisfied from the cache, the peer contacts the origin server to fetch the object, instead of forwarding it yet again.

When a large file is requested from CoBlitz, it generates a stream of requests for chunks of the file, and passes these requests to CoDeeN. These requests are spread across the CoDeeN nodes, which will either serve them from cache, or will forward them to the origin server. Replies from the origin server are cached at the CoDeeN nodes, and are returned to the original requestor. The details of the process are explained in our earlier work [9].

Our initial expectation for these systems was that CoDeploy would be used by PlanetLab researchers to

*Current contact: UC Irvine Computer Science Department. Work performed while a summer intern at Princeton

deploy and synchronize their experiments across nodes, while CoBlitz would be used for distributing content to the public, such as CD-ROM images. What we have found is that CoBlitz HTTP interface is quite simple to use, and can easily be integrated into deployment scripts or other infrastructure. As a result, we have seen individual researchers, other PlanetLab-based services, such as Stork [12] and PLuSH [10], and even our own group using CoBlitz to deploy and update files on PlanetLab.

This change in expected usage is significant for our design decisions, because it affects both the caching behavior as well as the desired goals. If the user population is large and the requested files are spread across a long period of time, high aggregate throughput, possibly at the expense of individual download speed, is desirable. At the same time, even if the number of copies fetched from the origin server is not minimal, the net benefit is still large.

In comparison, if the user population is mostly PlanetLab researchers deploying experiments, then many factors in the usage scenario change: the total number of downloads per file will be on the order of the number of nodes in PlanetLab (currently 583), all downloads may start at nearly the same time, the download latency becomes more important than the aggregate capacity, and extra fetches from the origin server reduces the benefits of the system.

Our goal in this work is to examine CoBlitz's design in light of its user population, and to make the necessary adjustments to improve its behavior in these conditions. At the same time, we want to ensure that the original audience for CoBlitz, non-PlanetLab users, will not be negatively affected. Since CoBlitz and CoDeploy share the same infrastructure, we expect that CoDeploy users will also see a benefit.

3 Observations & Redesign

In this section we discuss CoBlitz's behavior, the origins of the behavior, and what changes we made to address them. We focus on three areas: peering policies, reducing origin load, and reducing latency bottlenecks.

3.1 Peering

Background – When CoBlitz sends a stream of requests for chunks of a file into CoDeeN, these requests are dispersed across that CoDeeN node's peers, so the quality of CoDeeN's peering decisions can affect CoBlitz's performance. When CoDeeN's deployment was expanded from only North American PlanetLab nodes to all PlanetLab nodes, its peering strategy was changed such that each node tries to find the 60 closest peers within a 100ms round-trip time (RTT). The choice of using at most 60 peers was so that a once-per-second heartbeat could cycle through all peers within a minute, without generating too much background traffic. While techniques such as gossip [14] could reduce this traffic, we wanted to keep the pairwise measurements, since we were also interested in link health in addition to node status. Any heartbeat aggregation scheme might miss links between all pairs of peers. The 100ms cutoff was to reduce noticeable lag in interactive settings, such as Web browsing. In parts of the world where nodes could not find 20 peers within 100ms, this cutoff is raised to 200ms and the 20 best peers are selected. To avoid a high rate of change in the peer sets, hysteresis was introduced such that a peer was replaced only if another node showed consistently better RTTs.

Problem – To our surprise, we found that nodes at the same site would often have relatively little overlap between their peer lists, which could then have negative impacts on our consistent hashing behavior. The root of the problem was a high variance in RTT estimates being reinforced by the hysteresis. CoDeeN used application-level UDP "pings" in order to see application response time at remote nodes, and the average of a node's last 4 pings was used to determine its RTT. In most cases, we observed that at least one of the four most recent pings could be significantly higher than the rest, due to scheduling issues, application delays, or other non-network causes. Whereas standard network-level pings rarely show even a 10% range of values over short periods, the application-level pings routinely vary by an order of magnitude. Due to the high RTT variances, nodes were picking a very random subset of the available peers. The hysteresis, which only allowed a peer to be replaced if another was clearly better over several samples, then provided significant inertia for the members of this initial list – nodes not on the list could not maintain stable RTTs long enough to overcome the hysteresis.

Redesign – Switching from an *average* application-level RTT to the *minimum* observed RTT (an approach also used in other systems [3, 5, 11]) and increasing the number of samples yielded significant improvement, with application-level RTTs correlating well with ping time on all functioning nodes. Misbehaving nodes still showed large application-level minimum RTTs, despite having low ping times. The overlap of peer lists for nodes at the same site increasing from roughly half to almost 90%. At the same time, we discovered that many intra-PlanetLab paths had very low latency, and restricting the peer size to 60 was needlessly constrained. We increased this limit to 120 nodes, and issued 2 heartbeats per second. Of the nodes regularly running CoDeeN, two-thirds tend to now have 100 or more peers.

3.2 Reducing Origin Load

Background – When many nodes simultaneously download a large file via CoBlitz, the origin server will receive many requests for each chunk, despite the use of consistent hashing algorithms [13] designed to have multiple nodes direct requests for the same chunk to the same peer. In environments where each node will only download the

file once (such as software installs on PlanetLab), the relative benefit of CoBlitz drops as origin load increases.

Problem – When we originally tested using 130 North American nodes all downloading the same file, each chunk was downloaded by 15 different nodes on average, thereby reducing the benefit of CoBlitz to only 8.6 times that of contacting the origin directly. This problem stemmed from two sources: divergence in the peer lists, and the intentional use of multiple peers. CoBlitz's use of multiple peers per chunk stems from our earlier measurements indicating that it produced throughput benefits for cache hits [9]. However, increasing peer replication is a brute-force approach, and we are interested in determining how to do better from a design standpoint. The peer list divergence issue is more subtle – even if peer lists are mostly similar, even a few differences between the lists can cause a small fraction of requests to be sent to "non-preferred" peers. These peers will still fetch the chunks from the origin servers, since they do not have the chunks. These fetches are the most wasteful, since the peer that gets them will have little re-use for them.

Redesign – To reduce the effects of differing peer lists without requiring explicit peer list exchange between nodes, we make the following observation: with consistent hashing, if a node receives a forwarded request, it can determine whether it concurs that it is the best node to handle the request. In practice, we can determine when a request seems to have been inappropriately forwarded to a node, and then send it to a more suitable peer. To determine whether a request should be forwarded again or not, the receiving node calculates the list of possible peers for this request via consistent hashing, as though it had received it originally. If the node is not one of the top candidates on the list, then it concludes that the request was sent from a node with a differing peer list, and forwards it along. Due to the deterministic order of consistent hashing, this approach is guaranteed to make forward progress and be loop-free. While the worst case is a number of hops linear in the number of peer groups, this case is also exponentially unlikely. Even so, we limit this approach to only one additional hop in the redirection, to avoid forwarding requests across the world and to limit any damage caused by bugs in the forwarding logic. Observations of this scheme in practice indicated that typically 3-7% of all chunks require an extra hop, so restricting it to only one additional hop appears sufficient.

3.3 Addressing Latency Bottlenecks

Background – Much of the latency in downloading a large file stems from a small subset of chunks that require much more time to download than others. The agent on each CoDeeN node that generates the stream of chunk requests is also responsible for timing the responses and retrying any chunks that are taking too long. A closer examination of the slow responses indicates that some peers are much more likely than others to be involved. These nodes result in lower bandwidth for all CoBlitz transfers, even if they may not impact aggregate capacity.

Problem – When many requests begin synchronously, many nodes will simultaneously send requests for the same chunk to the peer(s) handling that chunk, resulting in bursty traffic demands. Nodes with less bandwidth will therefore take longer to satify this bursty traffic, increasing overall latency. While random request arrivals are not as affected, we have a user population that will often check for software updates using `cron` or some other periodic tool, resulting in synchronized request arrival. Though the download agent does issue multiple requests in parallel to reduce the impact of slower chunks, its total download rate is limited by the slowest chunk in the download window. Increasing the window size only increases the buffering requirement, which is unappealing since main memory is a limited resource.

Redesign – We observe that a simple way to reduce latency is to avoid peers that are likely to cause it, rather than relying on the agent to detect slow chunks and retry them. At the same time, improvements in the retry logic of the download agent can help eliminate the remaining latency bottlenecks. We experimented with two approaches to reducing the impact of the slowest nodes – reducing their frequency in the consistent hashing algorithms, and eliminating them entirely from the peering lists. Based on our bandwidth measurements of the various peers, described in Section 4.1, we tested both approaches and decided that avoiding slow peers entirely is preferable to modifying the hashing algorithms to use them. We present a discussion of our modified algorithm, along with its benefits and weaknesses, in Section 3.4. We also opted to make our download agent slightly more aggressive, drawing on the approach used in LoCI [2]. Previously, when we decided a chunk was taking too long to download, we stopped the transfer and started a new one with a different peer. In the majority of cases, no data had begun returning on the slow chunks, so this approach made sense. We modified the download agent to allow the previous transfer to continue, and let the two transfers compete to finish. With this approach, we can be more aggressive about starting the retry process earlier, since any work performed by the current download may still be useful.

3.4 Fractional Highest Random Weight

While the standard approach for handling heterogeneous capacities in consistent hashing has been the use of virtual nodes [7], we are not aware of any existing counterpart for the Highest Random Weight (HRW) [13] hashing scheme used in CoDeeN. Our concern is that using virtual nodes increases the number of items needed in the hashing scheme, and the higher computational cost of HRW (N * logN or N * # replicas versus logN for consistent hashing) makes the resulting computational requirements

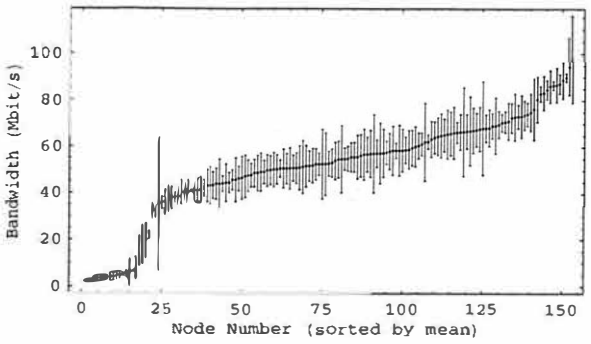


Figure 1: Mean node bandwidths & standard deviations

grow quickly. Our approach, Fractional HRW (F-HRW), does not introduce virtual nodes, and therefore requires only a modest amount of additional computation. HRW consists of three steps to assign a URL to a node: (1) hash the URL with every peer in the list, producing a set of hash values, (2) sort the peers according to these hash values, and (3) select the set of replicas with the highest values.

Our modification to HRW takes the approach of reducing the peer list based on the low-order bits of the hash value, such that peers are still included deterministically, but that their likelihood of being included on a particular HRW list is in proportion to their weight. For each peer, we assume we have a fractional weight in the range of 0 to 1, based on the expected capacity of the peer. In step (1), once we have a hash value for each node, we examine the low order bits (we arbitrarily choose 10 bits, for 1024 values), and only include the peer if the (low bits / 1024 < weight). We then sort as normal (or just pick the highest values via linear searches), as would standard HRW. Using the low-order bits to decide which peers to includes ensures that the decision to affect a peer is orthogonal to its rank in the sorted HRW list.

While F-HRW solves the issue of handling weights in HRW-based hashing, we find that it does not reduce latency for synchronized downloads. With F-HRW, the slow nodes do receive fewer requests *overall* versus the faster peers. However, for synchronized workloads, they still receive request bursts in short time frames, making them the download bottlenecks. For workloads where synchronization is not an issue, F-HRW can provide higher aggregate capacity, making it possibly attractive for some CDNs. However, when we examined the total capacity of the slowernodes in PlanetLab, we decided that the extra capacity from F-HRW was less valuable than the reduced latency from eliminating the slow peers entirely.

4 Evaluation

In this section, we describe our measurements of node bandwidths and of the various CoBlitz improvements.

Site (# nodes)	Node Avgs	Site Avg	Fastest
uoregon.edu (3)	2.46 - 2.66	2.59	4.63
cmu.edu (3)	3.50 - 3.95	3.67	5.74
csusb.edu (2)	3.93 - 4.21	4.07	6.76
rice.edu (3)	4.27 - 4.98	4.66	7.88
uconn.edu (2)	4.24 - 6.11	5.15	42.08

Table 1: Worst site bandwidths, measured in Mbps.

Site (# nodes)	Node Avgs	Site Avg	Slowest
neu.edu (2)	94.5 - 97.4	95.9	60.1
pitt.edu (1)	88.7	88.7	57.3
unc.edu (2)	84.6 - 87.1	85.9	66.1
rutgers.edu (2)	83.3 - 86.1	84.7	60.1
duke.edu (3)	80.5 - 89.9	84.2	59.6

Table 2: Best site bandwidths, measured in Mbps.

4.1 Measuring Node Bandwidths

To determine which peers are slow and should be excluded from CoBlitz, we perform continuous monitoring using a simple node bandwidth test. For each "edu" node on PlanetLab (corresponding to North American universities), we select the 10 closest peers, with no more than one peer per site, and synchronously start multiple TCP connections to the node from its peers. We measure the average aggregate bandwidth for a 30 second period, and repeat the test every 4 hours. Tests are run sequentially on the nodes, to avoid cross traffic that would occur with simultaneous tests. The results of 50 tests per node are shown in Figure 1. We show both the average bandwidth for each node, which ranges from 2.5 Mbps to 97.4 Mbps, as well as the standard deviation.

This straightforward testing reveals some interesting information regarding the characteristics of peak node bandwidths across these nodes: per-node bandwidth tends to be stable across time, all nodes at a site tend to be similar, and the disparities are quite large. While some nodes achieve very high bandwidths, we also observe a distinct group of poorly performing nodes that have significantly slower bandwidth speeds than the rest. There is a very large discrepancy between the best and worst sites, as outlined in Tables 1 and 2. We note that these properties are well-suited for our approach – slow nodes can be safely eliminated from consideration as peers via periodic measurements. In the event that fast nodes become slow due to congestion, the retry logic in the download agent can handle the change.

4.2 CoBlitz Improvements

To determine the effect of our redesign on CoBlitz, we measure client download times for both cached and uncached data, using various versions of the software. We isolate the impact of each design change, producing a set of seven different versions of CoBlitz. While these versions are intended to reflect our chronological changes,

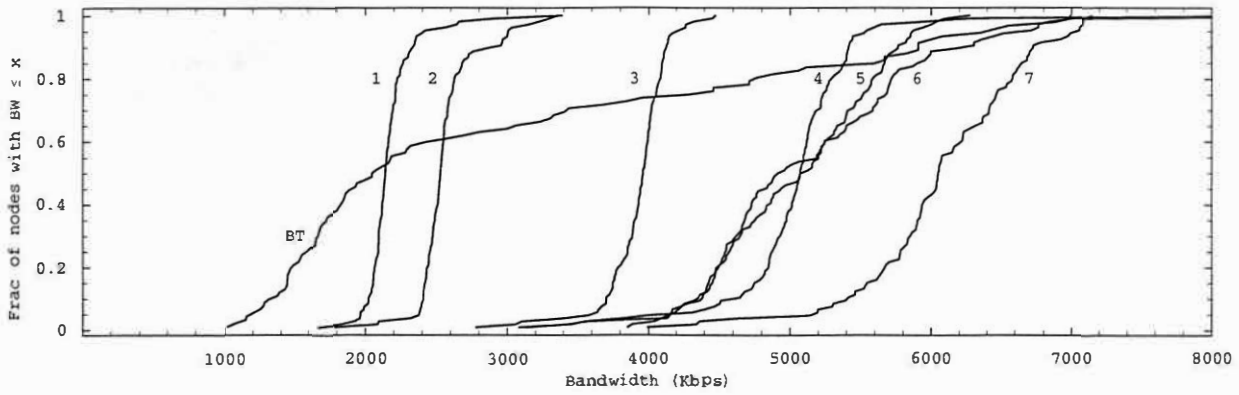


Figure 2: CDFs of mean node bandwidths for all design changes. Line numbers correspond to their entries in Table 3. Lines on the right have better bandwidths than lines on the left.

many of our changes occurred in overlapping steps, rather than a progression through seven distinct versions. In all scenarios, we use approximately 115 clients, running on North American university nodes on PlanetLab. All clients start synchronously, and download a 50 MB file located on a server at Princeton – once when the file is not cached by CoBlitz, and twice when it has already been downloaded once. We repeat each test three times and report average numbers.

Our seven test scenarios incrementally make one change at a time, and so that the final scenario represents the total of all of our modifications. The modifications are as follow: **Original** – CoBlitz as it started, with 60 peers, no exclusion of slow nodes, and the original download agent, **NoSlow** – exclude slow nodes (bandwidth < 20 Mbps) from being peers, **MinRTT** – replaces the use of average RTT values with minimum RTTs, **120Peers** – raises the limit of peers to 120, **RepFactor** – reduces the replication factor from 5 peers per chunk to 2 peers, **MultiHop** – bounces misdirected requests to more suitable peers, **NewAgent** – the more aggressive download agent. The download bandwidth for all clients on the uncached test is shown in Figure 2, and the summary data is shown in Table 3. We also include a run of BitTorrent on the same set of clients, for comparison purposes.

The most obvious change in this data is the increase in mean uncached bandwidth, from 2.1 to 6.1 Mbps, which improves our most common usage scenario. The CDFs show the trends more clearly – the design changes cause a rightward shift in the CDFs, indicating improved performance. The faster strategies also yield a wider spread of node bandwidths, but a wider spread of bandwidths is probably preferable to all nodes doing uniformly poorly. Not shown in the table is the average number of nodes requesting each chunk from the origin server, which starts at 19.0, drops to 11.5 once the number of peers is increased to 120, and drops to 3.8 after the MultiHop strategy is introduced. So, not only is the uncached bandwidth almost

#	Name	uncached	cached-1	cached-2
1	Original	2.1	5.8	6.6
2	NoSlow	2.5	5.3	6.8
3	MinRTT	3.9	6.7	6.9
4	120Peers	5.0	6.2	6.6
5	RepFactor	5.0	5.5	5.4
6	MultiHop	5.2	5.2	5.6
7	NewAgent	6.1	6.5	6.7
BT	BitTorrent	2.9	–	–

Table 3: Mean bandwidths in Mbps for the various redesign steps, for both uncached and cached downloads. Also included is the value for BitTorrent, for comparison

three times the original value, but load on the origin server is reduced to one-fifth its original amount.

This behavior also explains the trend in the cached bandwidths – the original numbers for the cached bandwidths are achieved through brute force, where a large number of peers are being contacted for each chunk. The initial reduction in cached bandwidth occurs because chunk downloads times become less predictable as the number of nodes serving each chunk drops. The cached bandwidths are finally restored using the more aggressive download agent, since more of the download delays are avoided by more tightly controlling retry behavior.

Note that our final version completely dominates our original version in all respects – not only is uncached bandwidth higher, but so is bandwidth on the cached tests. All of these improvements are achieved with a reduction of load to the origin server, so we feel confident that performance across other kinds of usage will also be improved. If CoBlitz traffic suddenly shifted toward non-PlanetLab users downloading large files from public Web sites, not only would they receive better performance than our original CoBlitz, but the Web sites would also receive less load.

System	# nodes	Median	Mean
CoBlitz cached	115	6.5	6.7
CoBlitz uncached	115	6.1	6.1
BitTorrent	115	2.0	2.9
Shark	185	1.0	
CoBlitz cached	41	7.3	8.1
CoBlitz uncached	41	7.1	7.4
BulletPrime	41		7.0

Table 4: Bandwidth results (in Mbps) for various systems at specified deployment sizes on PlanetLab. All measurements are for 50MB files, except for Shark, which uses 40MB.

5 Related Work

Due to space considerations, we cannot cover all related work in detail. The most obvious comparable system is BitTorrent [4], and our measurements show that we are twice as fast as it in these scenarios. Since BitTorrent was designed to handle large numbers of clients rather than high per-client performance, our results are not surprising. A more directly-related system is BulletPrime [8], which has been reported to achieve 7 Mbps when run on 41 PlanetLab hosts. In testing under similar conditions, CoBlitz achieved 7.4 Mbps (uncached) and 8.1 Mbps (cached) on average. We could potentially achieve even higher results by using a UDP-based transport protocol like Bullet's, but our current approach is TCP-friendly and is not likely to cause trigger any traffic concerns.

Finally, Shark [1], built on top of Coral [6], also performs a similar kind of file distribution, but uses the filesystem interface instead of HTTP. Shark's performance for transferring a 40MB file across 185 PlanetLab nodes shows a median bandwidth of 0.96 Mbps. Their measurements indicate that the origin server is sending the file 24 times on average in order to satisfy all 185 requests, which suggests that their performance may improve if they use techniques similar to ours to reduce origin server load. The results for all of these systems are shown in Table 4. The missing data for BulletPrime and Shark reflect the lack of information in the publications, or difficulty extracting the data from the provided graphs.

6 Conclusions

In this paper, we have shown how a detailed re-evaluation of several CDN design choices have significantly boosted the performance of CoBlitz. These design choices stemmed from two sources: our observations of our users' behavior, which differed substantially from what we had expected when launching the service, and from observing how our algorithms were behaving in practice, rather than just in theory. We believe we have learned two lessons that are broadly applicable: observe the workload your service receives to see if it can be optimized, and test the assumptions that underly your design once your service is deployed. For researchers working in content distribution networks or related areas, we believe that our experiences in hazards of peer selection and our algorithmic improvements (multi-hop, fractional HRW) may be directly applicable in other environments.

Acknowledgements

We would like to thank the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grants ANI-0335214 and CNS-0439842 and by Princeton University's Summer Undergraduate Research Experience (PSURE) program.

References

[1] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA. May 2005.

[2] M. Beck, D. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, M. Swany, S. Vadhiyar, and R. Wolski. Logistical computing and internetworking: Middleware for the use of storage in communication. In *3rd Annual International Workshop on Active Middleware Services (AMS)*, 2001.

[3] L. Brakmo, S. O'Malley, and L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the SIGCOMM '94 Symposium*, 1994.

[4] B. Cohen. Bittorrent. 2003. http://bitconjurer.org/BitTorrent.

[5] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[6] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, 2004.

[7] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, 1997.

[8] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of 2005 USENIX Annual Technical Conference*, 2005.

[9] K. Park and V. Pai. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proceedings of the First Workshop on Real, Large Distributed Systems(WORLDS '04)*, 2004.

[10] PLuSH. http://sysnet.ucsd.edu/projects/plush/.

[11] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[12] Stork. http://www.cs.arizona.edu/stork/.

[13] D. Thaler and C. Ravishankar. Using Name-based Mappings to Increase Hit Rates. In *IEEE/ACM Transactions on Networking*, volume 6, I. 1998.

[14] W. Vogels, R. van Renesse, and K. Birman. Using epidemic techniques for building ultra-scalable reliable communications systems. In *Workshop on New visions for Large-Scale Networks: Research and Applications*, Vienna, VA, 2001.

[15] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

The Julia Content Distribution Network

Danny Bickson* and Dahlia Malkhi**

Abstract— **Peer-to-peer content distribution networks are currently being used widely, drawing upon a large fraction of the Internet bandwidth. Unfortunately, these applications are not designed to be network-friendly. They optimize download time by using all available bandwidth. As a result, long haul bottleneck links are becoming congested and the load on the network is not well balanced.**

In this paper, we introduce the Julia content distribution network. The innovation of Julia is in its reduction of the overall communication cost, which in turn improves network load balance and reduces the usage of long haul links. Compared with the state-of-the-art BitTorrent content distribution network, we find that while Julia achieves slightly slower average finishing times relative to BitTorrent, Julia nevertheless reduces the total communication cost in the network by approximately 33%. Furthermore, the Julia protocol achieves a better load balancing of the network resources, especially over trans-Atlantic links.

We evaluated the Julia protocol using real WAN deployment and by extensive simulation. The WAN experimentation was carried over the PlanetLab wide area testbed using over 250 machines. Simulations were performed using the the GT-ITM topology generator with 1200 nodes. A surprisingly good match was exhibited between the two evaluation methods (itself an interesting result), an encouraging indication of the ability of our simulation to predict scaling behavior.

I. Introduction

Peer-to-peer content distribution networks are becoming widely utilized in today's Internet. The popular file sharing networks-—e.g. eMule, BitTorrent and KaZaA—have millions of online users. Current research shows that a large fraction of the Internet bandwidth is consumed by these applications [10]. Most existing solutions optimize download time while ignoring network cost, and put network load balance only as a secondary goal. As these networks become more popular, they consume increasing amounts of network bandwidth and choke the Internet. Eventually, their own performance deteriorates as a result of their success.

The approach we put forth in this paper takes network cost and balance into account from the outset. As in most existing solutions, the fundamental structure of our content delivery algorithm relies upon an origin node (or nodes) which stores a full copy of the content and which serves *pieces* of the content to a set of downloading clients.

The clients subsequently collaborate to exchange pieces among themselves. The novelty in our approach is that communication partners as well as the pieces exchanged with them, are chosen with the aim of reducing overall network usage, while at the same time, achieving fast download time. All of this is accomplished while maintaining tit-for-tat load sharing among participating nodes, which is crucial for incentivizing client participation.

Our consideration of the total network costs for content dissemination adopts similar goals to those considered by Demers et al. [7] in the context of gossip algorithms. Their spatial distribution algorithm is aimed to reduce the communication costs of disseminating a file in a network. Their basic idea is to prefer closer nodes: this is done by setting the cumulative probability of contacting a node to diminish exponentially with distance. Simulation results show that this technique significantly reduces the communication work, especially over long communication links. Our distance-aware node selection strategy closely follows this spatial distribution algorithm, with two important distinctions. First, our node selection policy changes over time, and adapts to the progress of the algorithm. Second, we vary the amount of data that is exchanged between nodes, and adapt it to the progress of the download.

The Julia algorithm has its roots in an earlier algorithm proposed by us in [2] for disseminating content over a structured hypercube topology. In this work, we propose a new algorithm to handle arbitrary network topologies, provide simulation results to confirm the design goals, and highlight real WAN deployment results over the PlanetLab [5] testbed.

Encouraging results are exhibited using two complementary evaluation methods, extensive simulations and a thorough PlanetLab testing over WAN. The two are compared against the BitTorrent [6] network under similar settings. Both simulation results and real planetary scale testing confirm our design goals: the network load balance over nodes and links shows improvement, while at the same time the communication cost is significantly

* School of Computer Science and Engineering, The Hebrew University of Jerusalem. daniel51@cs.huji.ac.il.

** Microsoft Research Silicon Valley and School of Computer Science and Engineering, The Hebrew University of Jerusalem. dalia@microsoft.com.

reduced. However, our system pays little in terms of running time.

The rest of this paper is structured as follows: In section II, we present the Julia algorithm. Next, we discuss protocol implementation in section III. In section IV, we report experimental results from both simulations and the PlanetLab test-bed. Finally, in section V, we present an improvement to the Julia algorithm and discuss its feasibility.

II. The Julia Algorithm

In this section, we introduce the Julia content distribution algorithm, which aims for the efficient transfer of large files (at least tens of megabytes).

One of the first design decisions we had to make in Julia is whether to use some predefined structured communication overlay. We favored an unstructured, constantly changing mesh, which is resilient against failures and requires no maintenance. In terms of data dissemination, having an unstructured mesh means that any pair of nodes can choose to exchange information. In the remainder of this section, we discuss the strategy for exchanging file pieces among nodes.

The main emphasis in the design of the Julia protocol is to reduce the overall communication cost of a file download, and to incur a balanced load on the network, without significantly impairing download completion time significantly. These design goals led to a probabilistic algorithm that determines which node to contact at every step. As in the spatial gossip algorithm [7], we prefer downloading from closer nodes whenever possible. However, the Julia node selection strategy is unique in that is adapts itself to the progression of the download. This adaptation is done roughly as follows. At the start of the download, the nodes do not have any information regarding the other nodes' bandwidths and latencies. Hence, each node will select nodes for pieces-exchange at random. As the download progresses, the nodes gossip and gather statistics about the network conditions. This knowledge is than used in order to contact progressively closer nodes.

In addition to the distance, we also vary the amount of data that is exchanged between interacting nodes: at the beginning of the download, we send a small number of pieces across each connection. As the download progresses — and as the quality of connections we utilize improves, we gradually increase the amount of data sent.

More formally, we have a file for download F, of size $|F| = k$ parts. Let x denote the number of pieces a node holds. The *progress* of a node is defined as $\frac{x}{k}$. The *distance* between nodes refers to the communication cost between them (the concrete parameters that determine the distance are an implementation matter; more on this in Section III. We use D_i to denote the maximal distance from node i to any other node.

The algorithm: Each node performs the selection of other nodes based on the following algorithm. Intuitively, we select nodes with an exponentially diminishing distance relative to the download progress.

Formally, we define $Q_i(d)$ as the set of nodes at a distance d or less from a node i. $Q_i(d)$ is known to i approximation only based upon the statistics gossiped during the download. Let node i have progress x/k. At each step of the algorithm, node i sets d to a value that reflects the download progress, using the exponential distribution formula $d = d(x/k) = D_i e^{-x/k}$. Node i then selects its next exchange partner uniformly at random from among all nodes in $Q_i(d)$, i.e., a node at distance up to d.

In this way, $x/k = 0$, at the start of the download, so that the initial selection is made from the entire universe of nodes $Q_i(D_i)$. When the download progress is about a halfway through, nodes from the closer group $Q_i(D_i e^{-1/2})$ are chosen. And so on, until close to the completion of the download, only very close nodes are selected.

III. The Implementation

We implemented a content distribution client in C++ based on the Julia algorithm. The client is implemented using a single thread server queue. The implementation code consists of approximately 15,000 lines of code, and uses TCP for the transport layer. To improve performance, the client maintains several (we used six) parallel connections. That is because larger number of TCP parallel connections result in poor download performance[1]. The decision of which node to contact next is made using the Julia algorithm.

One of the questions we had to answer when applying the Julia algorithm was how to calculate network distances. Different applications might have different views about distance. For example, streaming applications generally regard the communication latency as the distance, whereas file sharing applications usually consider the bandwidth as the main parameter to optimize. Other possible metrics include the number of hops or commonality of DNS suffixes. Additionally, local area links are cheaper to use than metropolitan links; metropolitan are cheaper than national links; and so on.

Our goal of reducing the communication cost dictates that we must use a combination of these parameters. We

[1]The same is done in the BitTorrent system where the actual downloading set of neighbors (out of the total neighbors set) is of size 4-5 [6]

took a similar approach for the Tulip routing overlay [1], and achieved a near optimal performance of routing. In Julia, we measure distance with a combination of bandwidth and latency. Note that latency is a good estimate of a link's physical length and, therefore, of its cost. However, we do not want to take only latency into account because this might interfere with the selection of high-bandwidth links.

Estimating distances in practice is another pragmatic challenge. The Julia client starts the data dissemination process with no knowledge of network conditions. Since we decided not to spend any extraneous bandwidth on active network probing, network conditions are discovered by passively monitoring the transfer rate of uploaded and downloaded file pieces. As information about network links is gathered, the client can apply the Julia algorithm to decide which neighbors to communicate with out of the known nodes. Note that this gradual process fits well with the Julia protocol, since early node selection in Julia inherently has great flexibility.

One important issue left out of the discussion so far is the strategy for selecting file pieces to send and receive. A Julia client maintains a bitmap of the pieces it has obtained so far. This bitmap is used in an exchange in order to ensure that only missing pieces are transmitted. Additionally, the client locally records the bitmaps that other clients have offered in previous rounds. This information is used for estimating the availability of file pieces throughout the network. As shown in [9], local estimation of file piece frequencies is a good approximation for global knowledge of the real frequencies.

Among those pieces missed by an exchange partner, our strategy is to send the rarest piece first. We adopted this strategy as a result of extensive experimentation with several selection policies [3].

IV. Experiments

A. The Simulation Method

The following are the performance measures we use in this paper: The download finishing *time* of a node is the time from the start of the download until the node has completely downloaded all file pieces. *Fair sharing* is the ratio between the number of file pieces the node forwards to the number of file pieces it receives. (In [4] this is called node stress.) Communication *work* is the product of file pieces traveled on a link and the link cost, summed over all the links. (In [4] this is called resource usage.)

Our simulation is done using a synchronous discrete event simulation we wrote, consisting of 3,000 lines of Java code. For the topology, we used the Georgia Tech topology generator (GT-ITM) [8] to create a transit-stub topology. We assigned stub-stub and stub-transit links bandwidth of 5 pieces per round, and transit-transit links bandwidth of 15 pieces per round. We used the link latencies, as created by the GT-ITM, to determine the link cost. The routing over the physical layer was done using Floyd all-pairs-shortest-path algorithm.

Out of the total of 600 physical nodes, we selected 200 random nodes to participate in the content distribution network. For each simulation, one source node was selected at random out of the 200 participating nodes. Each simulation was repeated at least 10 times and the results were averaged.

B. The PlanetLab Testing Method

Our PlanetLab test is done with a single source node storing the file in full, and about 250 nodes downloading simultaneously. The source node is used both for tracking other clients, and for retrieving pieces. Under a normal load, the source node provides a client that contacts it one data piece, the rarest, as well as a list of other nodes that previously connected to it. When the source node becomes overloaded, it stops serving pieces and provides only the list of nodes. After contacting the source node, clients exchange file pieces among themselves. We used file sizes of 30, 60 and 130Mb in our tests. Part size was set to 1/2Mb.

C. Preliminary Discussion of Results

It is enlightening to compare the simulation results to the real WAN experiments. The simulation environment is only a simplified approximation of a real system: nodes operate in synchronous rounds; the transmission of a piece is never disrupted; and all pieces sent in a round arrive before the start of the next round. Additionally, there are only two bandwidth categories, slow and fast. Reality is naturally more complex: No synchronization; heterogeneous machine capacities and diverse links; and there are node and network link failures, packet losses, congestion and unexpected delays.

Nevertheless, as we shall see below, a surprisingly good match is exhibited in our simulations of the PlanetLab settings. This is encouraging, as it suggests good prediction power for the simulation. The results below also indicate places where the simulation method may be improved for better accuracy.

D. Fair Sharing

Figure 1 provides a comparison of fair sharing in Julia and BitTorrent using both simulation and by deployment over PlanetLab. Overall, we observe a remarkably close match between the simulation results and the WAN measurements. This can be explained by the fact that

fair sharing is an algorithmic property of the protocol, and does not relate directly to bandwidth, or to the heterogeneity of the nodes.

The average fair sharing of both algorithms is a little less than one, which means that, on average, the network is load balanced. However, we can see that the Julia protocol provides a better load balancing of nodes, both for the simulation results and for PlanetLab. Surprisingly, WAN results show that, in practice, BitTorrent has a slightly higher fair sharing ratio than predicted. In contrast, the Julia client has a better fair sharing ratio than predicted (that is, closer to 1). We note that fair sharing is of immense importance for Peer-to-peer networks since it provides incentive to use the network.

E. Finishing Time

Figure 2 shows the completion times of our experiments. Here, the simulation and the PlanetLab results exhibit a slightly lower degree of matching than the Fair Sharing results above.

We speculate that the differences between the finishing times predicted by simulation and ones experienced through the PlanetLab tests are because the transit-stub model we use does not capture all of the PlanetLab network properties. For example, some machines in Brazil and Russia were behind lousy links, which made TCP perform poorly due to the slow start mechanism. Some of the machines are connected using ADSL, with asymmetric bandwidth properties, and had a narrow upload capability. Other machines were heavily loaded and performed poorly. Our simulation did not capture those network properties well.

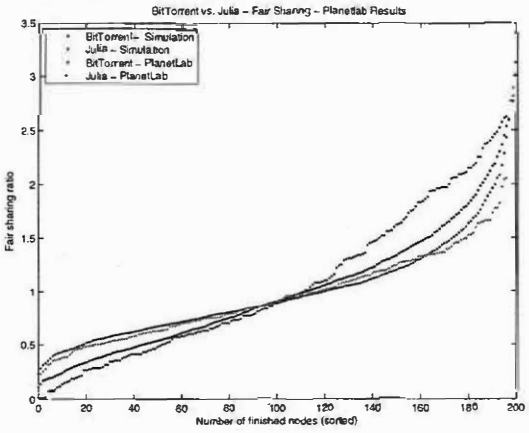


Fig. 1. *Comparison between simulation and WAN results of fair sharing (node stress). Fair sharing of 1 means that the node uploaded the same number of file pieces it downloaded from the network.*

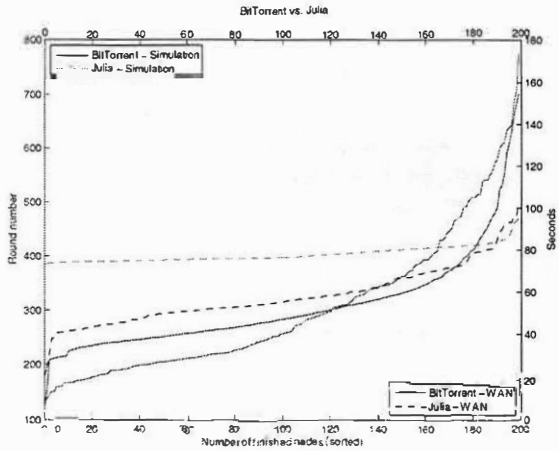


Fig. 2. *Finishing times for 200 nodes using simulation vs. PlanetLab results. The left Y-axes represent simulation rounds and the right Y-axes represent time in seconds. Note that the 50 slowest PlanetLab nodes were not shown in the graph because of their exponentially increasing finishing times, probably because of very slow or congested machines.*

F. Communication Cost

Our evaluation of the total communication cost is done only by simulation, since on PlanetLab, evaluating the costs incurred in practice is a challenging problem, mainly because there is no unified distance measurement. In our simulation, we used the link latencies as created by the transit-stub model for link costs.

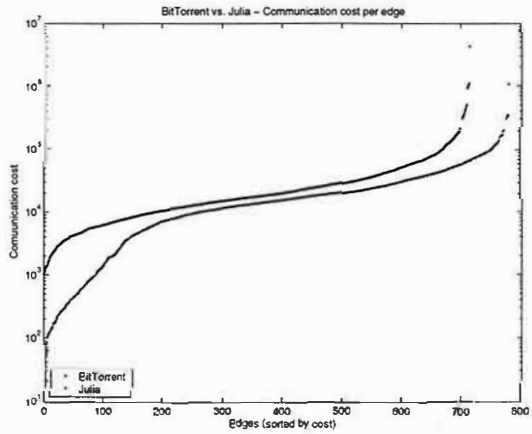


Fig. 3. *Total communication cost per edge in simulation. The average communication cost of transferring a file to a node in the Julia algorithm is reduced by 33% relative to the BitTorrent algorithm.*

Figure 3 shows simulation results of the communication costs per network link. The y-axis has a logarithmic cost scale. The x-axis presents the links ordered by their communication cost. Links with cost zero were removed from the graph. We can clearly see the advantage of

using Julia, resulting in a reduced network load. Simulation shows that the average communication cost of transferring the full file into each node is lowered by 33% relative to the BitTorrent algorithm.

We conducted an additional simulation, whose goal was to evaluate the load incurred on a costly trans-Atlantic link. To this end, we took two transit-stub networks of 600 nodes and connected their backbone using one link. The links in each network had bandwidth 5 pieces per round for transit-stub, and 15 pieces of round for stub-stub links. The trans-Atlantic link was assigned a bandwidth of 150 pieces per round. Two hundred nodes were selected at random to perform the overlay out of the total 1,200 physical nodes. We ran both the Julia and the BitTorrent algorithms to compare the number of file pieces traveled on the trans-Atlantic bottleneck link. As expected, this link was used in BitTorrent to transfer as much as four times the number of pieces relative to Julia. We conclude that the Julia algorithm has a potential not only to improve the network load balancing, but also in reducing traffic over the longer links.

V. Current Results and Future Directions

Based on the feedback we received from both the simulations and the PlanetLab testings, we are currently designing an improved version of the Julia algorithm. The crux of the improvement is as follows: In the basic Julia algorithm, neighbors are exchanged after the download of each piece. This might create a situation where a high bandwidth node nearby is exchanged for a slower node. We try to prevent this situation using a poker game strategy. The neighbors in our active download set are modeled as a hand of poker: we evaluate the upload performance of the neighbors, as we would evaluate our poker hand. Then, we allow the replacement of any neighbor with a performance below a certain threshold, similar to replacing any subset of poker cards out of our initial hand. We call our modified algorithm the Julia Poker variant.

This strategy is somewhat similar to the BitTorrent probing. In BitTorrent, each node probes for the bandwidth of one neighbor at a time, from among the fixed set of neighbors. If a probed node has a higher upload bandwidth, it is inserted into the active node set, and the lowest performing node is taken out of the active set. However, there are two major differences between the algorithms. In Julia, we allow the replacement of several nodes out of the active set and not just one at a time. Furthermore, the set of neighbors is not fixed. Nodes are selected from the complete network.

We believe our improved algorithm might work better in practice, since it is more flexible than the BitTorrent selection of nodes, while at the same time preserving the Julia algorithm properties of load balancing in the network. Preliminary simulation results confirming these predictions are shown in figure 4.

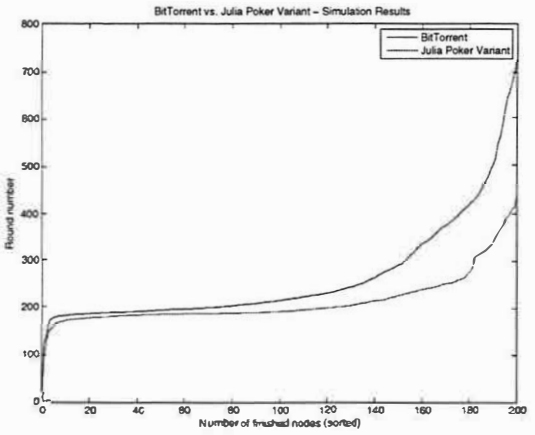


Fig. 4. *Finishing download times of BitTorrent vs. Julia Poker Variant using simulation.*

Acknowledgements We would like to thank Igal Ermanok for implementing the simulation.

References

[1] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, and S. Ron. Practical locality-awareness for large scale information sharing. 2005.

[2] D. Bickson, D. Malkhi, and D. Rabinowitz. Efficient large scale content distribution. In *The 6th Workshop on Distributed Data and Structures (WDAS'2004)*, July 2004.

[3] D. Bickson, D. Malkhi, and D. Rabonowitz. Locality aware content distribution. Technical Report TR-2004-52, 2004.

[4] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *In Proceedings of ACM SIGMETRICS, Santa Clara, CA, pp 1-12*, June 2000.

[5] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[6] B. Cohen. Incentives build robustness in bittorrent. In *Proceedings of P2P Economics Workshop*, 2003.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturighis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC, 1987.*

[8] K. C. Ellen W. Zegura and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM 1996, San Francisco, CA.*

[9] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *In proc. of INFOCOM 2005*, 2005.

[10] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. "file-sharing in the internet: A charachterization of p2p traffic in the backbone. In *Technical Report, University of California, Nov. 2003.*

Detecting Performance Anomalies in Global Applications*

Terence Kelly
Hewlett-Packard Laboratories
1501 Page Mill Road m/s 1125
Palo Alto, CA 94304 USA

Abstract

Understanding real, large distributed systems can be as difficult and important as building them. Complex modern applications that span geographic and organizational boundaries confound performance analysis in challenging new ways. These systems clearly demand new analytic methods, but we are wary of approaches that suffer from the same problems as the systems themselves (e.g., complexity and opacity).

This paper shows how to obtain valuable insight into the performance of globally-distributed applications without abstruse techniques or detailed application knowledge: Simple queueing-theoretic observations together with standard optimization methods yield remarkably accurate performance models. The models can be used for *performance anomaly detection*, i.e., distinguishing performance faults from mere overload. This distinction can in turn suggest both performance debugging tools and remedial measures.

Extensive empirical results from three production systems serving real customers—two of which are globally distributed and span administrative domains—demonstrate that our method yields accurate performance models of diverse applications. Our method furthermore flagged as anomalous an episode of a real performance bug in one of the three systems.

1 Introduction

Users and providers of globally-distributed commercial computing systems value application-level performance, because an unresponsive application can directly reduce revenue or productivity. Unfortunately, understanding application-level performance in complex modern distributed systems is difficult for several reasons. Today's commercial production applications are composed of numerous opaque software components running atop virtualized and poorly-instrumented physical resources. To make matters worse, applications are increasingly distributed across both geographical and organizational boundaries. Merely to collect in one place sufficient measurement data and knowledge of system design to support a detailed performance analysis is often very difficult in practice. Rapidly-changing application designs and configurations limit the useful life-span of an analysis once it has been performed.

For these reasons operators and administrators seldom analyze running production systems except in response to measurements (or user complaints) indicating unacceptably poor performance. The analyst's task is simplified if she can quickly determine whether the problem is due to excessive workload. If so, the solution may be as simple as provisioning additional resources for the application; if not, the solution might be to "reboot and pray." If such expedients are not acceptable and further analysis is required, knowing whether workload accounts for observed performance can guide the analyst's choice of tools: Ordinary overload might recommend resource bottleneck analysis, whereas degraded performance not readily explained by workload might suggest a fault in application logic or configuration. If different organizations manage an application and the systems on which it runs, quickly determining whether workload accounts for poor performance can decide who is responsible for fixing the problem, averting finger-pointing. In summary, *performance anomaly detection*—knowing when performance is surprising, given workload—does not directly identify the root causes of problems but can indirectly aid diagnosis in numerous ways.

This paper explores a simple approach to explaining application performance in terms of offered workload. The method exploits four properties typical of commercially-important globally-distributed production applications:

1. workload consists of request-reply *transactions*;
2. transactions occur in a small number of *types* (e.g., "log in," "browse," "add to cart," "check out" for an E-commerce site);
3. resource demands vary widely *across* but *not within* transaction types;
4. computational resources are adequately provisioned, so transaction response times consist largely of *service* times, not *queueing* times.

*Second Workshop on Real, Large Distributed Systems (WORLDS), San Francisco, 13 December 2005.
http://www.usenix.org/events/worlds05/

Data set	collection dates	duration	number of transactions	# trans'n types	$\sum_i \lvert e_i \rvert / \sum_i y_i$ OLS	LAR
ACME	July 2000	4.6 days	159,247	93	.2154	.1968
FT	Jan 2005	31.7 days	5,943,847	96	.1875	.1816
VDR	Jan 2005	10.0 days	466,729	37	.1523	.1466

Table 1: Summary of data sets and model quality.

We shall see that for applications with these properties, aggregate response time within a specified period is well explained in terms of transaction mix.

Our empirical results show that models of aggregate response time as a simple function of transaction mix have remarkable explanatory power for a wide variety of real-world distributed applications: Nearly all of the time, observed performance agrees closely with the model. The relatively rare cases where actual performance disagrees with the model can reasonably be deemed anomalous. We present a case study showing that our method identified as anomalous an episode of an obscure performance fault in a real globally-distributed production system.

Performance anomaly detection is relatively straightforward to evaluate and illustrates the ways in which our approach complements existing performance analysis methods, so in this paper we consider only this application of our modeling technique. Due to space constraints we do not discuss other applications, e.g., capacity planning and resource allocation.

2 Transaction Mix Models

We begin with a transaction log that records the type and response time of each transaction. We divide time into intervals of suitable width (e.g., 5 minutes for all experiments in this paper). For interval i let N_{ij} denote the number of transactions of type j that began during the interval and let T_{ij} denote the sum of their response times. We consider models of the form

$$y_i = \sum_j T_{ij} = \alpha_1 N_{i1} + \alpha_2 N_{i2} + \cdots \quad (1)$$

Note that no intercept term is present in Equation 1, i.e., we constrain the model to pass through the origin: aggregate response time must be zero for intervals with no transactions. For given vectors of model parameters a_j and observed transaction mix N_{ij} at time i, let

$$\hat{y}_i = f_{\vec{a}}(\vec{N}_i) = \sum_j a_j N_{ij} \quad (2)$$

denote the *fitted value* of the model at time i and let $e_i = y_i - \hat{y}_i$ denote the *residual* (model error) at time i. We define the *accuracy* of a model as a generalization of the familiar concept of relative error:

$$\text{normalized aggregate error} \equiv \frac{\sum_i |e_i|}{\sum_i y_i} \quad (3)$$

We say that a model of the form given in Equation 1 is *optimal* if it minimizes the figure of merit in Equation 3. We shall also report the *distribution* of residuals and scatterplots of $(y,\hat{y})$ pairs for our models. (The coefficient of multiple determination R^2 cannot be used to assess model quality; it is not meaningful because Equation 1 lacks an intercept term [17, p. 163].)

To summarize, our methodology proceeds through the following steps: 1) obtain parameters a_j by fitting the model of Equation 1 to a data set of transaction counts N_{ij} and response times T_{ij}; 2) feed transaction counts N_{ij} from *the same data set* into Equation 2 to obtain fitted values $\hat{y}_i$; 3) compare fitted values $\hat{y}_i$ with observed values y_i to assess model accuracy; 4) if the $\hat{y}_i$ agree closely with the corresponding y_i for most time intervals i, but disagree substantially for some i, deem the latter cases anomalous. We emphasize that we do *not* divide our data into "training" and "test" sets, and that our goal is *not* to forecast future performance. Instead, we retrospectively ask whether performance can be explained well in terms of offered workload throughout most of the measurement period. If so, the rare cases where the model fails to explain performance may deserve closer scrutiny.

Numerous methods exist for deriving model parameters a_j from data. The most widely-used procedure is ordinary least-squares (OLS) multivariate regression, which yields parameters that minimize the sum of squared residuals $\sum_i e_i^2$ [17]. Least-squares regression is cheap and easy: it is implemented in widely-available statistical software [18] and commercial spreadsheets (e.g., MS Excel). However it can be shown that OLS models can have arbitrarily greater normalized aggregate error than models that minimize Equation 3, and therefore we shall also compute the latter. Optimal-accuracy model parameters minimize the sum of absolute residuals $\sum_i |e_i|$. The problem of computing such parameters is known as "least absolute residuals (LAR) regression." LAR regression requires solving a linear program. We may employ general-purpose LP solvers [15] or specialized algorithms [4]; the computational problem of estimating LAR regression parameters remains an active research area [11].

Statistical considerations sometimes recommend one or another regression procedure. For instance, OLS and

LAR provide maximum-likelihood parameter estimates for different model error distributions. Another important difference is that LAR is a *robust* regression procedure whereas OLS is not: A handful of outliers (extreme data points) can substantially influence OLS parameter estimates, but LAR is far less susceptible to such distortion. This can be an important property if, for instance, faulty measurement tools occasionally yield wildly inaccurate data points. In this paper we shall simply compare OLS and LAR in terms of our main figure of merit (Equation 3) and other quantities of interest.

Intuitively, for models that include all transaction types j and for data collected during periods of extremely light load, parameters a_j represent typical *service* times for the different transaction types. Interaction effects among transactions are not explicitly modeled, nor are *waiting* times when transactions queue for resources such as CPUs, disks, and networks. Our ongoing work seeks to amend the model of Equation 1 with terms representing waiting times. This is not straightforward because the multiclass queueing systems that we consider are much harder to analyze than single-class systems [5] (classes correspond to transaction types). As we shall see in Section 3, the severe simplifying assumptions that we currently make do preclude remarkable accuracy.

Well-known procedures exist for simplifying models such as ours, but these must be used with caution. The number of transaction types can be inconveniently large in real systems, and a variety of refinement procedures are available for reducing in a principled way the number included in a model [17]. When we reduce the number of transaction types represented, however, parameters a_j no longer have a straightforward interpretation, and negative values are often assigned to these parameters. On the other hand, the reduced subset of transaction types selected by a refinement procedure may represent, loosely speaking, the transaction types most important to performance. Model refinement therefore provides an application-performance complement to procedures that automatically identify utilization metrics most relevant to performance [12]. We omit results on model refinement due to space limitations.

Measuring our models' accuracy is easy, but evaluating their usefulness for performance anomaly detection poses special challenges. If a model is reasonably accurate in the sense that observed performance y_i is close to the fitted value $\hat{y}_i$ for most time intervals i, why should we regard the relatively rare exceptions as "anomalous" or otherwise interesting? To address this question we model data collected on systems with known performance faults that occur at known times and see whether the model *fails* to explain performance during fault episodes.

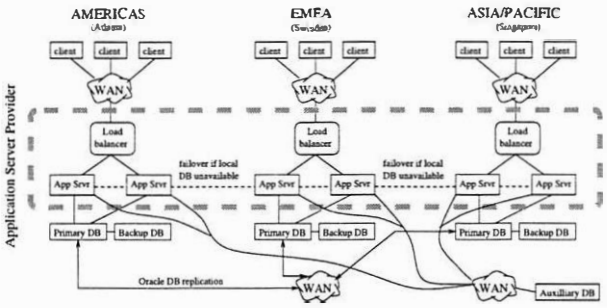


Figure 1: The globally-distributed "FT" application.

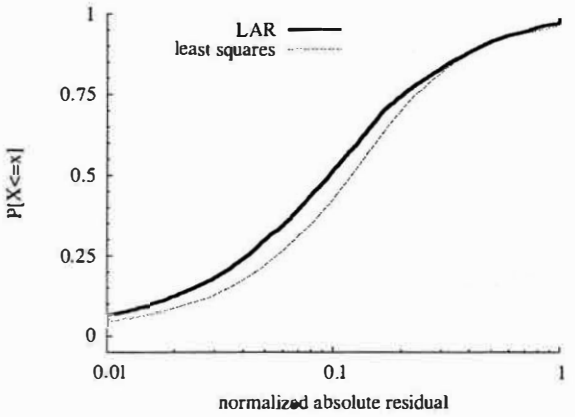


Figure 2: Cumulative distribution of $|e_i|/y_i$, FT data.

3 Empirical Evaluation

We evaluate the method of Section 2 using three large detailed data sets collected on real production systems. The first, which we call "ACME," was collected in July 2000 on one of several servers comprising a large Web-based shopping system; see Arlitt *et al.* for a detailed workload characterization [2]. The other two, which we call "FT" and "VDR," were collected in early 2005 on two globally-distributed enterprise applications serving both internal HP users and external customers. Cohen *et al.* provide a detailed description of FT [13]; VDR shares some features in common with FT but has not been analyzed previously. One noteworthy feature common to both FT and VDR is that different organizations are responsible for the applications and for the application-server infrastructure on which they run. Figure 1 sketches the architecture of the globally-distributed FT application; a dashed rectangle indicates managed application servers.

Table 1 describes our three data sets and presents summary measures of model quality for least-squares and LAR parameter estimation. Our figure of merit from Equation 3, $\sum_i |e_i| / \sum_i y_i$, shows that the models are quite accurate. In all cases, for LAR regression, normalized

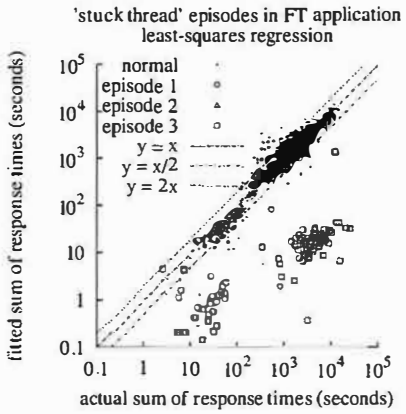


Figure 3: Scatterplot of y_i vs. $\hat{y}_i$, FT data.

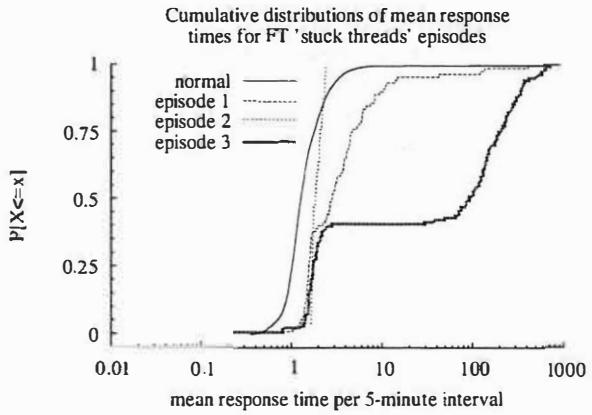


Figure 4: CDFs of mean response times during "stuck threads" episodes.

aggregate error ranges from roughly 15% to under 20%. Least-squares regression yields slightly worse models by this measure; it increases $\sum_i |e_i|$ by 3.2%–9.5% for our data. Figure 2 shows the cumulative distribution of absolute residuals normalized to y, i.e., the distribution of $|e_i|/y_i$, for the FT data and both regression procedures. The LAR model is wrong by 10% or less roughly half of the time, and it is almost never off by more than a factor of two. The figure also shows that LAR is noticeably more accurate than least-squares.

A scatterplot of fitted vs. observed aggregate response times offers further insight into model quality. Figure 3 shows such a plot for the FT data and OLS regression. Plots for LAR regression and other data sets are qualitatively similar: Whereas aggregate response times y range over several orders of magnitude, in nearly all cases fitted values $\hat{y}$ differ from y by less than a factor of two. A small number of points appear in the lower-right corner; these represent time intervals whose observed aggregate response times were far larger than fitted model values. For our data sets, the reverse is very rare, and very few points appear in the upper-left corner. Such points might indicate that transactions are completing "too quickly," e.g., because they quickly abort due to error.

As the FT data of Figure 3 was being collected, there occurred several episodes of a known performance fault that was eventually diagnosed and repaired. This fault, described in detail in [13], involved an application misconfiguration that created an artificial bottleneck. An important concurrency parameter in the application server tier, the maximum number of simultaneous database connections, was set too low. The result was that queues of worker threads waiting for database connections in the app server tier grew very long during periods of heavy load, resulting in excessively—and anomalously—long transaction response times. FT operators do not know precisely when this problem occurred because queue lengths, waiting times, and utilization are not recorded for finite database connection pools and other "soft" resources. However the admins gave us rough estimates that allow us to identify three major suspected episodes, shown with special points in Figure 3.

The most remarkable feature of the figure is that *false positives are extremely rare*: Data points for "normal" time intervals are almost never far from the $y = x$ diagonal and nearly all large discrepancies between y and $\hat{y}$ occur during suspected performance fault episodes. Unfortunately, false negatives do seem evident in the figure: Of the three suspected performance fault episodes, only episode 3 (indicated by open squares) appears far from $y = x$; most points corresponding to episodes 1 and 2 lie near the diagonal. Has our method failed to detect performance anomalies, or does the problem reside in our inexact conjectures regarding when episodes occurred? Figure 4 suggests the latter explanation. This figure shows the distributions of *average* (as opposed to *aggregate*) transaction response times for four subsets of the FT data: normal operation and the three alleged performance fault episodes. Figure 4 shows that episode 3—the one that stands out in Figure 3—has far higher mean response times than the other two episodes.

Several explanations are possible for our results. One possibility is that the problem did in fact occur during all three alleged episodes, and that our proposed anomaly detection method identifies only the most extreme case. Another possibility is that alleged episodes 1 and 2 were not actual occurrences of the problem. Based on how the alleged episodes were identified, and based on the large difference between episode 3 and the other two in Figure 4, the latter explanation seems more likely. (In a similar vein, Cohen *et al.* report that an episode of this problem on a host not analyzed here was initially misdiagnosed [13].) For our ongoing work we hope to an-

alyze systems with sporadic performance faults whose episodes are known with greater certainty. Data on such systems is hard to obtain, but it is required for a compelling evaluation of the proposed method.

4 Discussion

Section 3 shows that the very simple transaction mix performance models of Section 2 have remarkable explanatory power for real, globally-distributed production systems; they furthermore sometimes flag subtle performance bugs as anomalous. We would expect our technique to work well for any system that approximately conforms to the simplifying assumptions enumerated in Section 1: Workload consists of transactions that fall into a small number of types; service times vary less within types than across types; and resources are adequately provisioned so that service times dominate response times. This section discusses limitations inherent in our assumptions, the usefulness of the proposed method, and extensions to broaden its applicability.

We can identify plausible scenarios where our assumptions fail and therefore our method will likely perform poorly. If workload is moderately heavy relative to capacity, queueing times will account for an increasing fraction of response times, and model accuracy will likely suffer. We would also expect reduced accuracy if service times are inter-dependent across transaction types (e.g., due to resource congestion). For instance, "checkout" transactions may require more CPU time during heavy browsing if the latter reduces CPU cache hit rates for the former.

On the positive side, our method does not suffer if transactions are merely numerous, internally complex, or opaque. Furthermore it may flag as anomalous situations where problems are actually present but our simplifying assumptions are *not* violated. For instance, it can detect cases where transactions complete "too quickly," e.g., because they abort prematurely. Finally, our method can be used to detect anomalies in real time. At the close of every time window (e.g., every five minutes) we simply fit a model to all available data (e.g., from the previous week or month) and check whether the most recent data point is anomalous. LAR and OLS regressions may be computed in less than one second for the large data sets of Table 1.

Our ongoing work extends the transaction mix model of Equation 1 with additional terms representing queueing time. A naïve approach is simply to add resource utilization terms as though they were transaction types. Our future work, however, will emphasize more principled ways of incorporating waiting times, based on queueing theory. Perhaps the most important aspect of our ongoing work is to validate our methods on a wider range of real, large distributed systems. Testing model accuracy requires only transaction types and response times, which are relatively easy to obtain. However to verify that performance *anomalies* reported by our models correspond to performance *bugs* in real systems requires reliable information about when such bugs occurred, and such data is difficult to obtain.

5 Related Work

Researchers have proposed statistical methods for performance anomaly detection in a variety of contexts. Chen *et al.* [10] and Kiciman & Fox [16] use fine-grained probabilistic models of software component interactions to detect faults in distributed applications. Ide & Kashima analyze time series of application component interactions; their method detected injected faults in a benchmark application serving synthetic workload [14]. Brutlag describes a far simpler time-series anomaly detection method [6] that has been deployed in real production systems for several years [7]. Our approach differs in that it exploits knowledge of the transaction mix in workload and does not employ time series analysis; it is also far simpler than most previous methods.

If a performance problem has been detected and is not due to overload, one simple remedial measure is to re-start affected application software components. Candea & Fox argue that components should be designed to support deliberate re-start as a normal response to many problems [8]. Candea *et al.* elaborate on this theme by proposing fine-grained rebooting mechanisms [9].

On the other hand, if workload explains poor performance, a variety of performance debugging and bottleneck analysis tools may be applied. Barham *et al.* exploit detailed knowledge of application architecture to determine the resource demands of different transaction types [3]. Aguilera *et al.* and Cohen *et al.* pursue far less knowledge-intensive approaches to detecting bottlenecks and inferring system-level correlates of application-level performance [12, 1]. Cohen *et al.* later employed their earlier techniques in a method for reducing performance diagnosis to an information retrieval problem [13]. The performance anomaly detection approach described in this paper may help to inform the analyst's choice of available debugging tools.

Queueing-theoretic performance modeling of complex networked services is an active research area. Stewart & Shen predict throughput and mean response time in such services based on component placement and performance profiles constructed from extensive benchmarking [19]. They use a single-class M/G/1 queueing expression to predict response times. Urgaonkar *et al.* describe a sophisticated queueing network model of multi-tier applications [20]. This model requires rather exten-

sive calibration, but can be used for dynamic capacity provisioning, performance prediction, bottleneck identification, and admission control.

6 Conclusions

We have seen that very simple transaction mix models accurately explain application-level performance in complex modern globally-distributed commercial applications. Furthermore, performance faults sometimes manifest themselves as rare cases where our models *fail* to explain performance accurately. Performance anomaly detection based on our models therefore appears to be a useful complement to existing performance debugging techniques. Our method is easy to understand, explain, implement, and use; an Apache access log, a bit of Perl, and a spreadsheet suffice for a bare-bones instantiation. Our technique has no tunable parameters and can be applied without fuss by nonspecialists; in our experience it *always* works well "out of the box" when applied to real production systems.

More broadly, we argue that a principled synthesis of simple queueing-theoretic insights with an accuracy-maximizing parameter estimation procedure yields accurate and versatile performance models. We exploit only limited and generic knowledge of the application, namely transaction types, and we rely on relatively little instrumentation. Our approach represents a middle ground between knowledge-intensive tools such as Magpie on the one hand and nearly-knowledge-free statistical approaches on the other. Our future work explores other topics that occupy this interesting middle ground, including extensions of the method described here.

7 Acknowledgments

I thank Alex Zhang and Jerry Rolia for useful discussions of queueing theory, and statistician Hsiu-Khuern Tang for answering many questions about LAR regression and other statistical matters. Sharad Singhal, Jaap Suermondt, Mary Baker, and Jeff Mogul reviewed early drafts of this paper and suggested numerous improvements; WORLDS reviewers supplied similarly detailed and valuable feedback. The anonymous operators of the ACME, FT, and VDR production systems generously allowed researchers access to measurements of their system, making possible the empirical work of this paper. Researchers Martin Arlitt, Ira Cohen, Julie Symons, and I collected the data sets. Fereydoon Safai provided access to the linear program solver used to compute LAR parameters. Finally I thank my manager, Kumar Goswami, for his support and encouragement.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. SOSP*, pages 74–89, Oct. 2003.

[2] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Trans. on Internet Tech*, 1(1):44–69, Aug. 2001.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. OSDI*, pages 259–272, Dec. 2004.

[4] I. Barrodale and F. Roberts. An improved algorithm for discrete LI linear approximations. *SIAM Journal of Numerical Analysis*, 10:839–848, 1973.

[5] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 1998.

[6] J. Brutlag. Aberrant behavior detection in time series for network service monitoring. In *USENIX System Admin. Conf. (LISA)*, pages 139–146, Dec. 2000.

[7] J. Brutlag. Personal communication, Mar. 2005.

[8] G. Candea and A. Fox. Crash-only software. In *Proc. HotOS-IX*, May 2003.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Proc. OSDI*, Dec. 2004.

[10] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proc. NSDI*, Mar. 2004.

[11] K. L. Clarkson. Subgradient and sampling algorithms for l_1 regression. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 257–266, 2005.

[12] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. OSDI*, Oct. 2004.

[13] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proc. SOSP*, Oct. 2005.

[14] T. Ide and H. Kashima. Eigenspace-based anomaly detection in computer systems. In *Proc. SIGKDD*, Aug. 2005.

[15] ILOG Corporation. *CPLEX and related software documentation*. `http://www.ilog.com`.

[16] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, Spring 2005.

[17] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. Irwin, fourth edition, 1996.

[18] The R statistical software package, Apr. 2005. `http://www.r-project.org/`.

[19] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proc. NSDI*, pages 71–84, 2005.

[20] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. ACM SIGMETRICS*, pages 291–302, June 2005.

Bridging Local and Wide Area Networks for Overlay Distributed File Systems

Michael Closson and Paul Lu
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
{closson|paullu}@cs.ualberta.ca

Abstract

In metacomputing and grid computing, a computational job may execute on a node that is geographically far away from its data files. In such a situation, some of the issues to be resolved are: First, how can the job access its data? Second, how can the high latency and low bandwidth bottlenecks of typical wide-area networks (WANs) be tolerated? Third, how can the deployment of distributed file systems be made easier?

The Trellis Network File System (Trellis NFS) uses a simple, global namespace to provide basic remote data access. Data from any node accessible by Secure Copy can be opened like a file. Aggressive caching strategies for file data and metadata can greatly improve performance across WANs. And, by using a bridging strategy between the well-known Network File System (NFS) and wide-area protocols, the deployment is greatly simplified.

As part of the Third Canadian Internetworked Scientific Supercomputer (CISS-3) experiment, Trellis NFS was used as a distributed file system between high-performance computing (HPC) sites across Canada. CISS-3 ramped up over several months, ran in production mode for over 48 hours, and at its peak, had over 4,000 jobs running concurrently. Typically, there were about 180 concurrent jobs using Trellis NFS. We discuss the functionality, scalability, and benchmarked performance of Trellis NFS. Our hands-on experience with CISS and Trellis NFS has reinforced our design philosophy of layering, overlaying, and bridging systems to provide new functionality.

1 Introduction

When should a system be significantly redesigned? Or, should a more evolutionary approach be taken? Our experience with the Trellis Network File System (Trellis NFS) contributes a data point in support of an evolutionary and layered approach to distributed file systems.

In metacomputing and grid computing, a computational job may execute on a node that is geographically far away from its data files. For proper virtualization and transparency, some kind of remote data access system or distributed file system is required to provide the jobs with access to their data. Historically, the solutions have ranged from explicit stage-in and stage-out of the data [19, 1], to full-fledged distributed file systems [17].

Although it is quite common with batch schedulers (and other systems) to expect the user to explicitly move the data before the job is started (i.e., stage-in) and after the job is completed (i.e., stage-out), it is a substantial burden on the user. In particular, the user has to know in advance all of the data required by the job, which is error-prone. Also, depending on the specific application, the user has to know the algorithm for how the application maps command-line arguments and configuration files to job-specific names for input files and (even more problematically) for output files. Of course, one of the advantages of a real file system is that the application itself can generate, name, and gain access to files as it needs to. Admittedly, when things are perfect, the *de facto* stage-in/stage-out model does work, but a file system is more transparent and more functional.

Full-fledged distributed file systems, including the Andrew File System (AFS), are powerful systems. But, although AFS, and other systems, have many features to deal with performance issues across wide-area networks (WANs), it is not common to see any distributed file system deployed across WANs. The reasons include: First, not every site necessarily uses the same distributed file system. Second, not every site uses the same security model or system. Third, it can be difficult to arrange and maintain a common administrative policy across independent sites. Even when the technical difficulties can be solved, the social issues can veto a common distributed file system across administrative domains.

In the context of metacomputing (and grid computing), the common case scenario is a virtualized compu-

tational resource (aka metacomputer or grid) that spans different administrative domains as well as different geographical locations. The common case scenario is that all sites will not be running the same distributed file system nor the same security system. One solution to the problem of different administrative domains is to require that all participating sites must adopt a new infrastructure, such as the Grid Security Infrastructure (GSI) [5] for security. GSI has functionality, scalability, and robustness advantages. But, there are significant social (and political) reasons why imposing a common infrastructure might be difficult.

A different approach, taken by the Trellis Project [16], is to layer new functionality over selected, existing infrastructure. Whenever possible, the most common wide area and local area protocols and systems should be bridged and overlayed instead of replaced with new systems. We use the term "overlay" to refer to a layering of new functionality without significantly changing the semantics or implementation of the lower layers. On the one hand, maintaining the existing infrastructure makes it harder to make radical changes and gain (hopefully) commensurate radical improvements in functionality and performance. On the other hand, layering and overlaying are classic strategies that allow for easier deployment (especially across administrative domains) and compatibility with existing systems and applications.

However, the experience of the Trellis Project is that the overlay strategy works in practice. Our design objectives have been reinforced by our experience with the system, especially:

1. As much as possible, new functionality should be implemented at the user-level, instead of requiring superuser privileges and administrative intervention. When deploying systems across administrative domains, the need for superuser intervention is a significant liability. When unavoidable, superuser intervention should be as minimal and familiar as possible. For example, Trellis NFS does require a superuser to create the per-NFS-client machine mount point.

2. Existing systems, such as Secure Shell/Secure Copy and the Network File System (NFS), are flexible and robust enough to form the basis for new WAN-based systems.

3. The challenge lies in how to integrate the different components of an overlay system without having to replace the components. Interfaces and mechanisms for cooperation between systems can solve most problems.

2 The Trellis Project

The Trellis Project is attempting to create a software infrastructure to support *overlay metacomputing*: user-level aggregations of computing resources from different administrative domains. The research problems addressed by Trellis include scheduling policies [18, 14], wide area security [13], file system design [6], and new interfaces between different metacomputing components and applications.

In contrast to grid computing [9], overlay metacomputing requires minimal support from systems administrators. Trellis is implemented at the user-level and adds a thin layer of software over widely-deployed systems such as Secure Shell, Secure Copy, and NFS. It is an open research question as to whether a radical redesign of client-server (and peer-to-peer) computing (e.g., service-oriented architecture, new application programming interfaces (API), new software toolkits), as advocated by grid computing, is required or if an evolutionary and overlay approach will work equally well.

At least in the focused application domain of high-performance computing (HPC), Trellis and the overlay metacomputing approach has had some significant demonstrations of functionality and scalability. A series of experiments, dubbed the Canadian Internetworked Scientific Supercomputer (CISS), have used the Trellis system to solve real scientific problems, and aggregate thousands of processors and many administrative domains. With each subsequent experiment, the Trellis system has evolved with new functionality and redesigns based on the lessons learned. From 2002 to 2004, the experiments were:

1. CISS-1 [19]: November 4, 2002. The Trellis system aggregated 1,376 processors, within 18 administrative domains, at 16 different institutions across Canada, from the West Coast to the East Coast. A computational chemistry experiment (using MOLPRO) involving chiral molecules was computed over a continuous 24-hour production run. Over 7,500 jobs and over 3 CPU-years worth of computation were completed.

   CISS-1 proved that a global HPC job scheduler could be implemented entirely at the user-level, without the need for a job broker or resource discovery, via a pull-based model known as *placeholder scheduling* [18]. The most significant achievement of CISS-1 was the ability to aggregate 18 administrative domains, where each domain was only required to provide a normal, user-level account. When relying on the cooperation of different institutions, the fewer the requirements to participate, the more likely it is for them to agree.

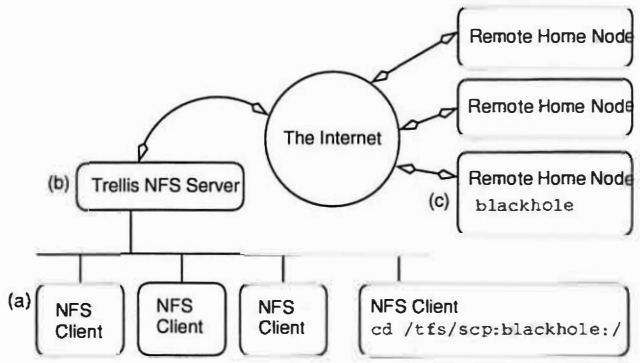


Figure 1: Trellis NFS. These components are (a) the NFS client(s), (b) the NFS server and (c) remote data storage server.

2. CISS-2: December 23, 2002 to January 2, 2003. The Trellis system aggregated hundreds of processors to complete two application workloads. A molecular dynamics problem was computed using GROMACS and a physics problem was computed using custom software.

   CISS-2 proved the usability of the Trellis software over an extended run and for multiple workloads.

3. CISS-3: September 15 to 17, 2004. Thousands of jobs were completed in the ramp-up to CISS-3 from April to September 2004. Then, in a 48-hour production run, over 15 CPU-years of computation were completed, using over 4,100 processors, from 19 universities and institutions. At its peak, over 4,000 jobs were running concurrently, including 180 jobs using the new distributed file system, Trellis NFS. Two different, unmodified binary applications were used: GROMACS and CHARMM.

   CISS-3 demonstrated the functionality and scalability of the Trellis NFS system. CISS-3 was also a proof-of-concept for the new Trellis Security Infrastructure (TSI) [13]. For CISS-3, TSI was developed to provide user-level, secure, scalable, single sign-on functionality for both interactive and background jobs.

3 Trellis NFS

The focus of this paper is the Trellis NFS file system. In providing a file system, the Trellis metacomputing system has specific advantages over the stage-in/stage-out approach. First, the application itself can name and access files on demand (e.g., generate filenames by adding suffixes to command-line arguments). Second, the system (and not the user) is responsible for data movement. Third, aggressive caching strategies can be implemented in a transparent manner.

The basic architecture of Trellis NFS is shown in Figure 1. Note that the NFS client is unmodified. Therefore, unmodified binary applications can use Trellis NFS. A similar strategy was used by the PUNCH Virtual File System [8], although with different WAN transport strategies, different mapping strategies between users and accounts, and different security mechanisms.

Running plain NFS over a WAN is not practical for several reasons. One reason is that the NFS protocol uses short, synchronous messages. Due to this, WAN latency renders NFS unsuitable. We traced the execution of a single invocation of the MAB60 (Modified Andrew Benchmark) [11] benchmark over NFS and counted 106,181 RPC calls which averaged 889 bytes in length for requests and 156 bytes in length for responses. Using the `netperf` [12] tool we measured a request-response rate of 4,537 transactions per second (TPS) for a 100Mbps Ethernet and 17 TPS for the optical network between the University of Alberta and the University of New Brunswick.

The Trellis NFS server is based on Linux's UNFSD server [20]. The server runs at the user-level, but a systems administrator must create the volume mount points on the NFS clients. The mount points can be shared among all processes running on the NFS client. This system administrator involvement is a small but unavoidable violation of Trellis' user-level strategy. By integrating the UNFSD server with the Trellis File System library [21], the modified UNFSD server can access files on any WAN node accessible by Secure Copy and serve the files to unmodified NFS clients with NFS semantics. The global names used by Trellis NFS (see Figure 1, NFS Client), such as `/tfs/scp:blackhole.westgrid.ca:/data/file1`, are recognizable as Secure Copy inspired names, and are known as Secure Copy Locators (SCL).

The Trellis NFS server uses aggressive data caching to overcome the effects of running a file system over a high-latency network. A local disk (or file system) is used as a cache, called the Trellis cache, in a manner similar to Web caches, although the data in Trellis can be read-write files. Files are copied into the Trellis cache on demand. Because an NFS server cannot know when a client has closed a file, a timeout feature is used to schedule the copy-back of the file to the home node, but we are experimenting with the Trellis scheduler telling Trellis NFS when the copy-back should occur. Thus, Secure Copy is the WAN protocol for accessing files and the normal NFS RPC protocol is what Trellis NFS uses to serve NFS clients on the LAN.

For HPC workloads, the common case is whole-file access and caching, and thus Secure Copy is used. For sparse updates of files, it is also possible to transparently use `rsync`-over-`ssh` to move the data in an efficient way, but our HPC workloads do not (in general) benefit from `rsync`'s optimizations.

Through TSI [13], Trellis NFS supports multiple users. As with normal NFS, security on Trellis NFS's LAN-client side is based on the (implicit) security of the local nodes and local user identities. For WAN security, TSI and Trellis NFS uses Secure Shell's agents and public-key authentication (and authorization). Trellis NFS security over a WAN is as secure as using Secure Shell to access remote accounts and, for example, using the Concurrent Version System (CVS) over Secure Shell to share a CVS repository between different users [3]. There are some practical and theoretical security problems with NFS, but NFS is still widely used. Rather than try to replace NFS, Trellis tries to bridge NFS and the WAN.

A newly developed Trellis SAMBA server provides the same functionality as Trellis NFS, but with SAMBA's [2] per-user authentication and the ability to detect when a file is closed. With both Trellis NFS and Trellis SAMBA, the bridging strategy is the key design decision. We focus here on Trellis NFS because of its relative maturity and track record with CISS-3.

3.1 Trellis NFS During CISS-3

As discussed above, we used the Trellis NFS server as part of CISS-3. The CISS-3 experiment included two applications: First, GROMACS [15] is a molecular dynamics simulator. Second, CHARMM [4] is a macromolecular simulator written in Fortran.

The Trellis NFS *server* was used in two administrative domains participating in the CISS-3 experiment. Each server handled multiple NFS clients on a local LAN. The first site was a cluster at the University of Alberta; a 20-node, 40-processor Linux cluster. The second site was a cluster at the University of New Brunswick; an 80-node, 160-processor Linux cluster (Table 1). Between the two domains, 200 jobs were using Trellis NFS at the peak, but the average was 180 jobs. The input and output data for the GROMACS application (i.e., first home node) were stored on a server at the University of Calgary. The input and output data for the CHARMM application were stored on a server at Simon Fraser University (i.e., second home node). Thus, for CISS-3, Trellis NFS was running across nodes in a total of four administrative domains, in three Canadian provinces, and separated by thousands of kilometres of WAN.

Statistic	Average per Hour
Number of NFS Clients (2 jobs per client)	68 to 73
LAN Data Written	200 MB
LAN Data Read	75 MB
LAN RPCs	40,000

Table 1: Trellis NFS Summary for New Brunswick site during CISS-3. All statistics are per-hour.

We are analyzing our trace data from CISS-3. Table 1 is based on an initial analysis of the LAN traffic at the New Brunswick site. The Trellis NFS traffic is determined by the workload and applications themselves. Both applications, like many scientific applications, have a burst of read-only activity at job start-up time and a burst of write-only activity at job exit time. Very few of the LAN RPCs, which are mostly NFS `getattr` RPCs, result in WAN RPCs due to Trellis NFS's cache [6]. Trace data from the New Brunswick site shows one WAN `getattr` RPC for every 360 cache-served LAN `getattr` RPCs. For these workloads, the WAN read and write statistics are likely very similar to the LAN statistics, given the read-only, write-only patterns.

3.2 Micro-benchmark: Bonnie++

The Bonnie++ micro-benchmark [7] is normally used to evaluate local disk subsystem performance. We use Bonnie++ primarily as a standardized workload to do an online measurement of Trellis NFS's performance. There are 3 stages in the Bonnie++ benchmark: write, read, and re-write. First, three 1 gigabyte files are created and written using the `write()` system call. Second, the 3 gigabytes of data is read back using the `read()` system call. Third, the 3 gigabytes of data is split into 16 KB pages; each page is read, modified and re-written (using `lseek()`).

Table 2 shows the throughput of the read, write and re-write tests, including (in line (c)) the additional MD5 computation and data transfer overheads required when

Configuration	Read	Write	Re-write
Local Disk	55.4 (2.8)	23.3 (0.36)	15.5 (0.32)
UNFSD (baseline)	24.1 (0.57)	22.1 (0.62)	7.6 (0.23)
(a) Trellis NFS over LAN	23.9 (0.79)	22.3 (0.54)	7.7 (0.13)
(b) Trellis NFS over WAN	24.4 (0.61)	22.5 (0.54)	7.6 (0.13)
(c) Trellis NFS over LAN; cold cache, flush cache	7.0 (2.87)	5.3 (2.15)	3.1 (1.23)

Table 2: Bonnie++ throughput. All results are in megabytes per second, averaged over 10 runs, standard deviation is in parentheses. Higher numbers are better. (a) and (b) are from NFS client's point of view, with warm caches and no data flush. (c) is for end-to-end performance, including a cache miss, flushing data to home node, and MD5 hash to check data integrity.

moving data in and out of cache. Although the overheads are a key part of Trellis NFS, the main bottleneck is the WAN itself and all distributed file systems would experience similar overheads since they would have the same WAN bottleneck. From the NFS client's point of view, the overheads can be overlapped with computation and amortized over a multi-hour run, so it is useful to also measure the performance from the NFS client's point of view (lines (a) and (b) of Table 2). The marginally better performance of Trellis NFS over WAN (line (b)) as compared to over a LAN (line (a)) is within measurement noise. Not surprisingly, local disk read performance is about 2.3 times faster then the NFS configurations. We include the local disk for perspective. The write test shows all four test configurations have almost equal performance, when working out of cache, which is the common case after the initial start of most HPC applications.

The key conclusion is that, once Trellis NFS has the file data in its cache, the performance of Trellis NFS is comparable to UNFSD. But, when the cache is cold, the performance is bottlenecked by the WAN for data movement. Future work will look at reducing the number of cold misses in the Trellis cache, but the current situation is both functional and reasonable, especially when compared to the UNFSD baseline.

4 Concluding Remarks

Trellis NFS bridges the LAN-based NFS protocol with WAN-based protocols, like Secure Copy, to provide a distributed file system that is relatively easy to deploy because it is (mostly) at the user-level and it is overlayed on top of existing systems. As with the other elements of the Trellis Project, Trellis NFS has been tested in a variety of Canada-wide experiments, known as CISS, that serve to provide empirical evaluations and feedback to the design of the Trellis system.

For future work, we plan to address some of the shortcomings of Trellis NFS and the Trellis system in general. First, the cache consistency model of Trellis NFS is, perhaps, too simple. Currently, there is no locking mechanism for multiple writers of the same file to synchronize their actions. The last job to "close" the file will overwrite all of the updates made by previous jobs. Fortunately, in the HPC application domain, it is rare for applications to actively read-write the same file. Jobs are often well-partitioned into independent, job-specific input (e.g., `config.1-2-1.input`) and output (e.g., `config.1-2-1.output`) files. However, other application domains and file systems (e.g., the append operation of the Google File System [10]) may require some form of file consistency control or multiple-writer semantics.

Second, as previously mentioned, we are implementing a version of the Trellis File System using SAMBA [2] instead of NFS. Among the advantages of using SAMBA include the ability to have per-user authentication to the SAMBA server (instead of NFS's per-client machine security model), avoiding superuser-created mount points (since *some* SAMBA clients allow unprivileged users to create mount points), and the ability of the SAMBA server to see when files are closed, which will make it easier to develop cache consistency strategies. SAMBA's stackable Virtual File System (VFS) mechanism is also a cleaner implementation technique, as compared to modifying a user-level NFS server directly. Of course, the Trellis SAMBA server is consistent with the bridging strategy of the Trellis NFS implementation.

5 Acknowledgments

This research is supported by the Natural Science and Engineering Research Council of Canada (NSERC), SGI, the Alberta Science and Research Authority (ASRA), and Sun Microsystems. Thank you to the Trellis and CISS teams. Thank you to the referees and to Adriana Iamnitchi, our shepherd, for their insightful and helpful suggestions.

References

[1] Portable Batch System. http://www.openpbs.org.

[2] SAMBA. http://www.samba.org.

[3] D. J. Barrett and R. E. Silverman. *SSH, the Secure Shell: The Definitive Guide.* O'Reilly, 2001.

[4] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4:187–217, 1983.

[5] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

[6] M. Closson. The Trellis Network File System. Master's thesis, Department of Computing Science, University of Alberta, 2004.

[7] R. Coker. The Bonnie++ benchmark. http://www.coker.com.au/bonnie++/.

[8] R.J. Figueiredo, N.H. Kapadia, and J.A.B. Fortes. The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid. In *10th IEEE Int'l Symposium on High Performance Distributed Computing*, San Francisco, California, 2001.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, 2002. Open Grid Service Infrastructure WG, Global Grid Forum.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, Bolton Landing, NY, U.S.A., October 19–22 2003.

[11] J.H. Howard, M.L. Kazar, S.G.Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[12] R. Jones. Netperf. http://www.netperf.org/netperf/NetperfPage.html.

[13] M. Kan, D. Ngo, M. Lee, P. Lu, N. Bard, M. Closson, M. Ding, M. Goldenberg, N. Lamb, R. Senda, E. Sumbar, and Y. Wang. The Trellis Security Infrastructure: A Layered Approach to Overlay Metacomputers. In *The 18th International Symposium on High Performance Computing Systems and Applications*, 2004.

[14] N. Lamb. Data-Conscious Scheduling of Workflows in Metacomputers. Master's thesis, Department of Computing Science, University of Alberta, 2005.

[15] E. Lindahl, B. Hess, and D. van der Spoel. GROMACS 3.0: A package for molecular simulation and trajectory analysis. *J. Mol. Mod.*, 7:306–317, 2001.

[16] P. Lu. The Trellis Project. http://www.cs.ualberta.ca/~paullu/Trellis/.

[17] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(4):184–201, 1986.

[18] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 85–105, Edinburgh, Scotland, UK, July 24 2002. Also published in Springer-Verlag LNCS 2537 (2003), pages 205–228.

[19] C. Pinchak, P. Lu, J. Schaeffer, and M. Goldenberg. The Canadian Internetworked Scientific Supercomputer. In *17th International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 193–199, Sherbrooke, Quebec, Canada, May 11–14 2003.

[20] M. Shand, D. Becker, R. Sladkey, O. Zborowski, F. van Kempen, and O. Kirch. The Linux UNFSD Server. ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir/.

[21] J. Siegel and P. Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (Cluster 2002)*, pages 480–483, Chicago, Illinois, USA, September 23–36 2002.

Non-Transitive Connectivity and DHTs

Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica
New York University and University of California, Berkeley
mfreed@cs.nyu.edu, {karthik,srhea,istoica}@cs.berkeley.edu

1 Introduction

The most basic functionality of a distributed hash table, or DHT, is to partition a key space across the set of nodes in a distributed system such that all nodes agree on the partitioning. For example, the Chord DHT assigns each node a random identifier from the key space of integers modulo 2^{160} and maps each key to the node whose identifier most immediately follows it. Chord is thus said to implement the *successor* relation, and so long as each node in the network knows its predecessor in the key space, any node can compute which keys are mapped onto it.

An implicit assumption in Chord and other DHT protocols is that all nodes are able to communicate with each other, yet we know this assumption is unfounded in practice. We say a set of three hosts, A, B, and C, exhibit *non-transitivity* if A can communicate with B, and B can communicate with C, but A cannot communicate with C. As we show in Section 2, 2.3% of all pairs of nodes on PlanetLab exhibit transient periods in which they cannot communicate with each other, but in which they can communicate through a third node. These transient periods of non-transitivity occur for many reasons, including link failures, BGP routing updates, and ISP peering disputes (e.g., [15]).

Such non-transitivity in the underlying network is problematic for DHTs. Consider for example the Chord network illustrated in Figure 1. Identifiers increase from the left, so node B is the proper successor to key k. If nodes A and B are unable to communicate with each other, A will believe that C is its successor. Upon receiving a lookup request for k, A will return C to the requester. If the requester then tries to insert a document associated with k at node C, node C would refuse, since according to its view it is not responsible for key k.

While this example may seem contrived, it is in fact quite common. If each pair of nodes with adjacent identifiers in a 300-node Chord network (independently) has a 0.1% chance of being unable to communicate, then we expect that there is a $1-0.999^{300} \approx 26\%$ chance that *some* pair will be unable to communicate at any time. However, both nodes in such a pair have a 0.999^2 chance of being able to communicate with the node that most immediately precedes them both.

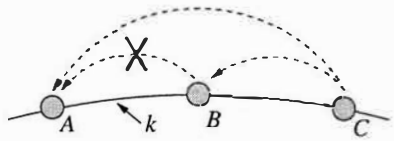


Figure 1: *Non-transitivity in Chord.* The dashed lines represent predecessor links.

Collectively, the authors have produced three independent DHT implementations: the Bamboo [20] implementation in OpenDHT [21], the Chord [25] implementation in *i*3 [24], and the Kademlia [13] implementation in Coral [9]. Moreover, we have run public deployments of these three DHTs on PlanetLab for over a year.

While DHT algorithms seem quite elegant on paper, in practice we found that a great deal of our implementation effort was spent discovering and fixing problems caused by non-transitivity. Of course, maintaining a full link-state routing table at each DHT node would have sufficed to solve all such problems, but would also require considerably more bandwidth than a basic DHT.[1] Instead, we each independently discovered a set of "hacks" to cover up the false assumption of full connectivity on which DHTs are based.

In this paper, we categorize the ways in which Bamboo, Chord, and Kademlia break down under non-transitivity, and we enumerate the ways we modified them to cope with these shortcomings. We also discuss application-level solutions to the problem. Many of these failure modes and fixes were quite painful for us to discover, and we hope that—at least in the short term—this work will save others the effort. In the longer term, we hope that by focusing attention on the problem, we will encourage future DHT designers to tackle non-transitivity head-on.

The next section quantifies the prevalence of non-transitivity on the Internet and surveys related work in this area. Section 3 presents a brief review of DHT terminology. Section 4 discusses four problems caused by non-transitivity in DHTs and our solutions to them. Finally, Section 5 concludes.

[1] For some applications, link-state routing may in fact be the right solution, but such systems are outside the scope of our consideration.

2 Prevalence of Non-Transitivity

The Internet is known to suffer from network outages (such as extremely heavy congestion or routing convergence problems) that result in the loss of connectivity between some pairs of nodes [3,16]. Furthermore, the loss of connectivity is often non-transitive; in fact, RON [3] and SOSR [11] take advantage of such non-transitivity—the fact that two nodes that cannot temporarily communicate with one another often have a third node that can communicate with them both—to improve resilience by routing around network outages.

Gerding and Stribling [10] observed a significant degree of non-transitivity among PlanetLab hosts; of all possible unordered three tuples of nodes (A,B,C), about 9% exhibited non-transitivity.[2] Furthermore, they attributed this non-transitivity to the fact that PlanetLab consists of three classes of nodes: Internet1-only, Internet2-only, and multi-homed nodes. Although Internet1-only and Internet2-only nodes cannot directly communicate, multi-homed nodes can communicate with them both.

Extending the above study, we have found that *transient* routing problems are also a major source of non-transitivity in PlanetLab. In particular, we considered a three hour window on August 3, 2005 from the all-pairs ping dataset [1]. The dataset consists of pings between all pairs of nodes conducted every 15 minutes, with each data point averaged over ten ping attempts.

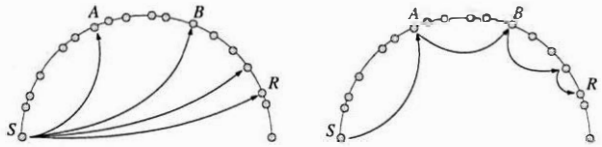
We counted the number of unordered pairs of hosts (A,B) such that A and B cannot reach each other but another host C can reach both A and B. We found that, of all pairs of nodes, about 5.2% of them belonged to this category over the three hour window. Of these pairs of nodes, about 56% of the pairs had persistent problems; these were probably because of the problem described above. However, the remaining 44% of the pairs exhibited problems intermittently; in fact, about 25% of the pairs could not communicate with each other only in one of the 15-minute snapshots. This suggests that non-transitivity is *not* entirely an artifact of the PlanetLab testbed, but also caused by transient routing problems.

3 DHT Background

Before moving on to the core of this paper, we first briefly review basic DHT nomenclature. We assume the reader has some familiarity with basic DHT routing protocols. For more information, see [13,23,25].

The DHT assigns every key in the identifier space to a node, which is called the *root* (or the *successor*) of the key. The main primitive that DHTs support is *lookup*, in which a node can efficiently discover a key's root. The lookup protocol greedily traverses the nodes of the DHT, progressing closer to the root of the key at each step.

[2] Li et al. [12] have later studied the effect of such non-transitivity on the robustness of different DHTs such as Chord and Tapestry.



(a) Iterative routing (b) Recursive routing

Figure 2: Two styles of DHT routing for source node S to perform a lookup that terminates at root node R.

Each node maintains a set of neighbors that it uses to route packets. Typically, such neighbors are divided into (a) *short links* chosen from the node's immediate neighborhood in the ID space to ensure correctness of lookups, and (b) *long links* chosen to ensure that lookups are efficient (e.g., take no more than $O(\log n)$ hops for a network with n nodes). In Chord and Bamboo, the set of short links is called the node's *successor list* and *leaf set*, respectively, and the long links are called *fingers* and *routing table entries*. While Kademlia uses a single routing table, one can still differentiate between its closest *bucket* of short links and farther buckets of long links.

DHT routing can be either *iterative* or *recursive* [8] (see Figure 2). Consider a simple example, in which source node S initiates a lookup for some key whose root is node R. In iterative routing, node S first contacts node A to learn about node B, and then S subsequently contacts B. In recursive routing, S contacts A, and A contacts B in turn.

Both routing techniques have different strengths. For example, recursive routing is faster than iterative routing using the same bandwidth budget [8, 19] and can use faster per-node timeouts [20]. On the other hand, iterative routing gives the initiating node more end-to-end control, which can be used, for instance, for better parallelization [13, 19]. We discuss the impact of both approaches in the following section.

4 Problems and Solutions

This section presents problems caused by non-transitivity in DHTs and the methods we use to mitigate them. We present these problems in increasing order of how difficult they are to solve.

4.1 Invisible Nodes

One problem due to non-transitivity occurs when a node learns about system participants from other nodes, yet cannot directly communicate with these newly discovered

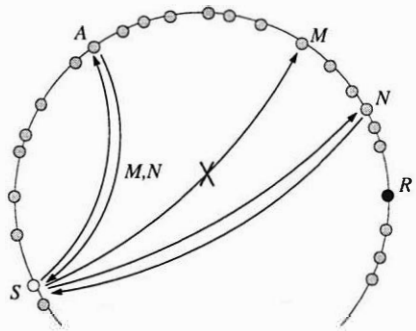


Figure 3: *Invisible nodes.* S learns about M and N from A while trying to route to R, but S has no direct connectivity to M. By sending lookup messages to M and N in parallel, S avoids being stalled while its request to M times out.

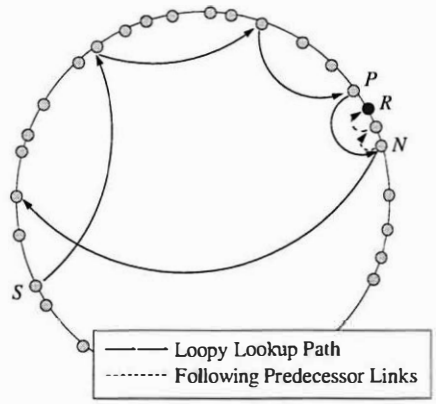


Figure 4: *Routing loops.* In *i*3-Chord, if a lookup passes by the correct successor on account of non-transitivity, a routing loop arises. The correctness of lookup can be improved in such cases by traversing predecessor links.

nodes. This problem arises both during neighbor maintenance and while performing lookups.

For example, assume that a node A learns about a potential neighbor B through a third node C, but A and B cannot directly communicate. We say that from A's perspective B is an *invisible node*. In early versions of both Bamboo and *i*3-Chord, A would blindly add B as a neighbor. Later, A would notice that B was unreachable and remove it, but in the meantime A would try to route messages through B.

A related problem occurs when nodes blindly trust failure notifications from other nodes. Continuing the above example, when A fails to contact B due to non-transitivity, in a naive implementation A will inform C of this fact, and C will erroneously remove B as a neighbor.

A simple fix for both of these problems is to prevent nodes from blindly trusting other nodes with respect to which nodes in the network are up or down. Instead, a node A should only add a neighbor B after successfully communicating with it, and A should only remove a neighbor with whom it can no longer directly communicate. This technique is used by all three of our DHTs.

Invisible nodes also cause performance problems during iterative routing, where the node performing a lookup must communicate with nodes that are not its immediate neighbors in the overlay. For example, as shown in Figure 3, a node S may learn of another node M through its neighbor A, but may be unable to directly communicate with M to perform a lookup. S will eventually time out its request to M, but such timeouts increase the latency of lookups substantially.

Three techniques can mitigate the effect of invisible nodes on lookup performance in iterative routing. First, a DHT can use virtual coordinates such as those computed by Vivaldi [7] to choose tighter timeouts. This technique should work well in general, although we have found that the Vivaldi implementations in both Bamboo and Coral are too inaccurate on PlanetLab to be of much use.[3]

Second, a node can send several messages in parallel for each lookup, allowing requests to continue towards the root even when some others time out. As shown in Figure 3, S can send lookup messages to M and N in parallel. This technique was first proposed in Kademlia [13].

Third, a node can remember other nodes that it was unable to reach in the past. Using this technique, which we call *a unreachable node cache*, a node S marks M as unreachable after a few failed communication attempts. Then, if M is encountered again during a subsequent lookup request, S immediately concludes that it is unreachable without wasting bandwidth and suffering a timeout.

OpenDHT and *i*3 both use recursive routing, but Coral implements iterative routing using the above approach, maintaining three parallel RPCs and a unreachable node cache.

4.2 Routing Loops

In *i*3-Chord, non-transitivity causes routing loops as follows. *i*3-Chord forwards a data packet to the root for a key k, which is the node whose identifier most immediately succeeds k in the circular key space. In Figure 4, let the proper root for k be R. Also, assume that P cannot communicate with R. A lookup routed through P thus skips over R to N, the next node in the key space with which P can communicate. N, however, knows its correct predecessor in the network, and therefore knows that it is

[3]We note, however, that neither of our Vivaldi implementations include the kinds of filtering used by Pietzuch, Ledlie, and Seltzer to produce more accurate coordinates on PlanetLab [17]: it is possible that their implementation would produce more accurate timeout values.

not the root for k. It thus forwards the lookup around the ring, and a loop is formed.

It should be noted that this problem does not occur in the original version of the Chord protocol, since a node does not forward a lookup request to the target node [25]; instead the predecessor of the target node returns the target node to the requester. However, this operation would introduce an extra RTT delay in forwarding an $i3$ packet, and still will not eliminate the problems created by non-transitive routing (see the example in Figure 1).

Bamboo and Kademlia avoid routing loops by defining a total ordering over nodes during routing. In these networks, a node A only forwards a lookup on key k to another node B if $|B-k| < |A-k|$, where "$-$" represents modular subtraction in Bamboo and XOR in Kademlia.

Introducing such a total ordering in $i3$-Chord is straightforward: instead of forwarding a lookup towards the root, a node can stop any lookup that has already passed its root. For example, when N receives a lookup for k from P, it knows something is amiss since $P < k < N$, but N is not the direct successor of k. An alternative mechanism for preventing loops would be to store a key on its predecessor node, rather than its successor node [6].

Stopping a lookup in this way avoids loops, but it is often possible to get closer to the root for a key by routing along predecessor links once normal routing has stopped. $i3$'s Chord implementation backtracks in this way. For example, the dashed lines from N back to R in Figure 4 show the path of the lookup using predecessor links. To guarantee termination when backtracking, once a packet begins following predecessor links it is never again routed along forward links.

4.3 Broken Return Paths

Often an application built atop a DHT routing layer wants to not only route to the root of a key but also to retrieve some value back. For example, it may route a put request to the root, in which case it expects an acknowledgment of its request in return. Likewise, with a get request, it expects to receive any values stored under the given key. In one very important case, it routes a request to join the DHT to the root and expects to receive the root's leaf set or successor list in return.

As shown in Figure 5, when a source S routes a request recursively to the root R, the most obvious and least costly way for R to respond is to communicate with S directly over IP. While this approach works well in the common case, it fails with non-transitivity; the existence of a route from S to R through the overlay does not guarantee the existence of the direct IP route back. We know of two solutions to this problem.

The first solution is to source route the message backwards along the path it traveled from S to R in the first

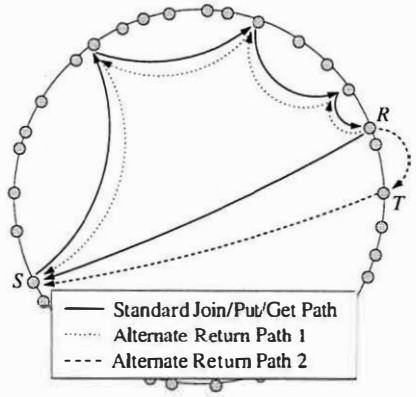


Figure 5: *Broken return paths.* Although S can route a put or get request to R through the overlay, there may be no direct IP route back from R to S. One alternative is to route the result back along the path taken from S to R; the other is to route through a random neighbor T.

place, as shown by the dotted line in Figure 5. Since each node along the path forwarded the message through a neighbor that had been responding to its probes for liveness, it is likely that this return path is indeed routable. A downside of this solution is that the message takes several hops to return to the client, wasting the bandwidth of multiple nodes.[4]

A less costly solution is to have R source route its response to S through a random member of its leaf set or successor list, as shown by the dashed line in Figure 5. These nodes are chosen randomly with respect to R itself (by the random assignment of node identifiers), so most of them are likely to be able to route to S. Moreover, we already know that R can route to them, or it would not have them as neighbors.

A problem with both of these solutions is that they waste bandwidth in the common case where R can indeed send its response directly to S. To avoid this waste, we have S acknowledge the direct response from R. If R fails to receive an acknowledgment after some timeout, R source routes the response back (either along the request path or through a single neighbor). This timeout can be chosen using virtual coordinates, although we have had difficulty with Vivaldi on PlanetLab as discussed earlier. Alternatively, we can simply choose a conservative timeout value: as it is used only in the uncommon case where R cannot route directly to S, it affects the latency of only a few requests in practice. Bamboo/OpenDHT routes back through a random leaf-set neighbor in the case of non-transitivity, using a timeout of five seconds.

[4] A similar approach, where R uses the DHT's routing algorithm to route its response to S's identifier, has a similar cost but a lower likelihood of success in most cases, so we ignore it here.

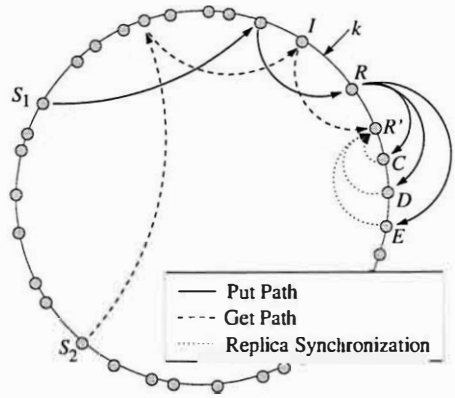


Figure 6: *Inconsistent roots.* A put from S_1 is routed to the root, R, which should replicate it on R',C,D. But since R cannot communicate with R', it replicates it on C–E instead. R' will later acquire a replica during synchronization with C–E.

We note that iterative routing does not directly suffer from this problem. Since S directs the routing process itself, it will assume R is down and look for an alternate root R' (i.e., the node that would be the root if R were actually down). Of course, depending on the application, R' may not be a suitable replacement for R, but that reduces to the inconsistent root problem, which we discuss next.

4.4 Inconsistent Roots

The problems we have discussed so far are all routing problems. In this section, we discuss a problem caused by non-transitivity that affects the correctness of the partitioning of the DHT key space.

Most DHT applications assume that there is only one root for a given key in the DHT at any given time. As shown in Figure 6, however, this assumption may be invalid in the presence of non-transitivity. In the figure, node R is the proper root of key k, but since R and R' cannot communicate, R' mistakenly believes it is the root for k. A lookup from S_1 finds the correct root, but a lookup from S_2 travels through node I, which also cannot communicate with R, and terminates instead at R'.

Prior work has explored the issue of multiple roots due to transient conditions created by nodes joining and leaving the overlay, but has not explored the effects of misbehavior in the underlying network [4].

Given a complete partition of the network, it is difficult to solve this problem at all, and we are not aware of any existing solutions to it. On the other hand, if the degree of non-transitivity is limited, the problem can be eliminated by the use of a consensus algorithm. The use of such algorithms in DHTs is an active area of research [14,22].

Nonetheless, consensus is expensive in messages and bandwidth, so many existing DHTs use a probabilistic approach to solving the problem instead. For example, FreePastry 1.4.1 maintains full link-state routing information for each leaf set, and a node is considered alive if any other member of its leaf set can route to it [2]. Once routability has been provided in this manner, existing techniques (e.g., [4]) can be used to provide consistency.

An alternative approach used by both DHash [5] and OpenDHT [18] is to solve the inconsistent root problem at the application layer. Consider the traditional put/get interface to hash tables. As shown in Figure 6, DHash sends a put request from S_1 for a key-value pair (k, v) to the r closest successors of k, each of which stores a replica of (k, v).[5] In the figure, R cannot communicate with R', and hence the wrong set of nodes store replicas.

To handle this case, as well as normal failures, the nodes in each successor list periodically synchronize with each other to discover values they should be storing (see [5, 18] for details). As shown in the figure, R' synchronizes with C–E and learns about the value put by S_1. A subsequent get request from S_2 which is routed to R' will thus find the value despite the non-transitivity.

Of course, if R' fails to synchronize with C–E between the put from S_1 and the get from S_2, it will mistakenly send an empty response for the get. To avoid this case, for each get request on key k, DHash and OpenDHT query multiple successors of k. For example, in the figure, R' would send the get request to C–E, and all four nodes would respond to S_2, which would then compile a combined response. This extra step increases correctness at the cost of increased latency and load; OpenDHT uses heuristics to decide when this extra step can be eliminated safely [19].

5 Conclusion

In this paper, we enumerated several ways in which naive DHT implementations break down under *non-transitivity*, and we presented our experiences in dealing with the problems when building and deploying three independent DHT-based systems—OpenDHT [21] that uses Bamboo [20], *i*3 [24] that uses Chord [25], and Coral [9] that uses Kademlia [13]. While we believe that the ultimate long-term answer to dealing with issues arising from non-transitivity is perhaps a fresh DHT design, we hope that, at least in the short term, this work will save others the effort of finding and fixing the problems we encountered.

[5] DHash actually stores erasure codes rather than replicas, but the distinction is not relevant to this discussion.

6 Acknowledgements

The authors would like to thank Frank Dabek, Jayanthkumar Kannan, and Sriram Sankararaman for their comments which helped to improve the paper.

References

[1] PlanetLab All-Pairs Pings. `http://pdos.lcs.mit.edu/~strib/pl_app/`.

[2] Freepastry release notes. `http://freepastry.rice.edu/FreePastry/README-1.4.1.html`, May 2005.

[3] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. SOSP*, 2001.

[4] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Dec. 2003.

[5] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[6] F. Dabek. Personal communication, Oct. 2005.

[7] F. Dabek, R. Cox, F. Kaahoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. SIGCOMM*, 2004.

[8] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. NSDI*, 2004.

[9] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. NSDI*, Mar. 2004.

[10] S. Gerding and J. Stribling. Examining the tradeoffs of structured overlays in a dynamic non-transitive network, 2003. Class project: `http://pdos.csail.mit.edu/~strib/docs/projects/networking_fall2003.pdf`.

[11] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *Proc. OSDI*, 2002.

[12] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating dht design tradeoffs under churn. In *Proc. INFOCOM*, 2005.

[13] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS*, 2002.

[14] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT-LCS, June 2005.

[15] D. Neel. Cogent, Level 3 in standoff over Internet access. TechWeb, Oct. 2005.

[16] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, U.C. Berkeley, 1997.

[17] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on PlanetLab. 2005.

[18] S. Rhea. *OpenDHT: A public DHT service*. PhD thesis, U.C. Berkeley, Aug. 2005.

[19] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proc. WORLDS*, Dec. 2005.

[20] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Tech. Conf.*, June 2004.

[21] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. SIGCOMM*, Aug. 2005.

[22] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT CSAIL, Dec. 2003.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Nov. 2001.

[24] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. SIGCOMM*, Aug. 2002.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM*, Aug. 2001.

Why it is hard to build a long-running service on PlanetLab

Justin Cappos and John Hartman
Computer Science Department
University of Arizona
Tucson, AZ, 85721
{justin,jhh}@cs.arizona.edu

Abstract

PlanetLab was conceived as both an experimental testbed and a platform for long-running services. It has been quite successful at the former, less so at the latter. In this paper we examine why. The crux of the problem is that there are few incentives for researchers to develop long-running services. Research prototypes fulfill publishing requirements, whereas long-running services do not. Several groups have tried to deploy *research services*, long-running services that are useful, but also novel enough to be published. These services have been generally unsuccessful. In this paper we discuss the difficulties in developing a research service, our experiences in developing a research service called Stork, and offer suggestions on how to increase the incentives for researchers to develop research services.

1 Introduction

Building a long-running service on PlanetLab[12] is difficult. PlanetLab has been used extensively to test and measure experimental research prototypes, but few long-running services are in wide-spread use. At the root of the problem is the "dual use" purpose of PlanetLab as originally conceived. On the one hand, PlanetLab is a research testbed, giving researchers access to numerous geographically-distributed nodes, realistic network behavior, and realistic client workloads. The bulk of PlanetLab activity to date has been of this sort. Most of the services that run on PlanetLab are short-lived *research prototypes*, developed as part of various research projects and funded by research funding agencies. As such, the prototypes exist to support experiments and produce publishable results. They must be sufficiently novel and must function only well enough to run the necessary experiments and collect the necessary results to validate the design.

On the other hand, PlanetLab is supposed to be a platform for deploying long-lived services, connecting researchers who want to produce these services with the users who want to use them. Presumably these services are based on earlier research prototypes, and incorporate novel features that the users find appealing. To date, very few of these long-lived services have been deployed and even fewer have come into wide-spread use.

The reasons for this lie in the lack of incentives for researchers to produce long-lived services. The original PlanetLab paper is notably silent on this subject: PlanetLab's dual purpose is touted as the most distinguishing characteristic of the PlanetLab approach to changing the Internet, but no blueprint is given for how this dual use will come to be. Most of the discussion revolves around how to support it, rather than how to make it a reality.

As a result, there are few long-lived services on PlanetLab. PlanetLab is designed and used by researchers for whom the reward structure provides little incentive to convert research prototypes into long-lived services. Research prototypes suffice for publication; since the reward for developing a new research prototype is quite high, and the reward for converting an existing prototype into a long-lived service is quite low, it is little wonder that PlanetLab is awash in prototypes while suffering a drought of services.

A potential middle ground is the *research service*, a long-lived service with sufficient research content to warrant publication. They are more interesting than simple services, and more available and reliable than research prototypes. Unfortunately, by spanning the gap between simple services and research prototypes, research services must meet the requirements of both worlds (Figure 1). They must have a research component so as to fit into the standard research reward structure, yet must not have any corner cases that make them unreliable. They must be long-lived, yet permit experimentation. There are several reasons why it is extremely difficult to build a successful research service:

- Research services must contain a research component that supports a research hypothesis. They must do something new and interesting, rather than established and mundane, as the reward system values novelty.
- Research services must rely on other research services. A research service need not be novel in all respects, but in those areas where it is not, it is expected to make use of the current state-of-the-art re-

	Research Prototypes	Research Services	Services
Reliability	Unimportant	Very Important	Very Important
Novelty	Very Important	Very Important	Unimportant
Research Interest	High	High	Low
Example Services	Bullet SHARK	Stork Bellagio CoBlitz	Sirius CoMon AppManager

Figure 1: *Characteristics of research services, services, and research prototypes.*

search results. If, for example, recent research has shown that reliable data transfer is best achieved using a particular technique, a research service that incorporates reliable data transfer will be expected to use that technique. That means that research services end up depending on one another. Since a well-known aphorism is to avoid having one's research depend on another research project, this is not a recipe for success.

- Research services are prone to instability. Research prototypes ignore the corner cases for a reason – they are difficult to handle and don't contribute to the research results. This means that a research service tends to have a relatively high bug/failure rate. Interaction between research services makes the problem worse. Each additional service introduces its own set of corner cases, increasing the size of the corners and decreasing overall stability.

For the remainder of the paper we describe the development cycle of a PlanetLab research service called *Stork*. We then discuss the issues and problems that cause research services to enter a downward spiral on PlanetLab, and conclude with suggestions on how to break the cycle by providing incentives for researchers to develop research services rather than research prototypes.

2 Stork

Stork is a PlanetLab research service that installs and maintains software for other services. A key problem facing PlanetLab services is the difficulty in installing software on a large set of nodes and keeping that software updated over a long period of time. Researchers need to quickly and efficiently distribute new package content to huge numbers of nodes, and do so in the face of network and node failures. In such an environment, nodes may miss software updates, however the correct software state must eventually be reached. In addition, software must be installed on the nodes efficiently. Each node runs hundreds of slices, many of which will install the same software. Having hundreds of copies of the same software on a node is not feasible; provisions must be made for sharing copies between slices.

Stork solves this software maintenance problem. Software installed using Stork is securely shared between slices, and a package is not downloaded if another slice on the same node already has it installed. Stork uses efficient transfer mechanisms such as CoBlitz[4] and BitTorrent[5] to transfer files. Stork has security features that allow developers to share a package on a repository without trusting each other or the package repository administrator.

Stork uses the functionality provided by other long-running services to enhance its capabilities. Stork uses Proper[10] to share and protect content, CoDeeN[19] and CoBlitz to transfer files, and AppManager[8] to deploy itself on every node.

We first give an overview of the Stork project throughout its development. We then discuss the specific problems that we encountered during different phases of development and how these problems are instances of the more general problems that plague research services.

2.1 Timeline

We present a timeline for Stork in Figure 2. The timeline illustrates the different phases of the project during development and deployment. Each period begins when the first line of code for a version was written. The resources, implementation and tool features for each version are also discussed.

Throughout the design and development of Stork we performed incremental roll-out and development. Our intention was to gradually build Stork's capabilities along with its user base, relying on user feedback to help define its development.

2.1.1 Plan-apt (Stork predecessor)

Plan-apt is a precursor to Stork that we developed for the purpose of installing and updating packages on PlanetLab nodes. Plan-apt is essentially a simple remote execution program with functionality tailored to package installation. It allows a single host to push package updates to many client slices. Unfortunately, since plan-apt required the user to start a daemon in each slice to be managed, the setup cost was unattractive.

We were frustrated by the inefficiencies of plan-apt with regard to setup, disk usage, and security. We decided to shelve remote execution to instead focus on a tool that securely shares package content between slices on a single node.

2.1.2 Stork (Alpha release)

We developed the first version of Stork to address plan-apt's shortcomings. This version uses apt to fetch packages and resolve dependencies, but provides extra security and disk space savings. It was mostly focused on

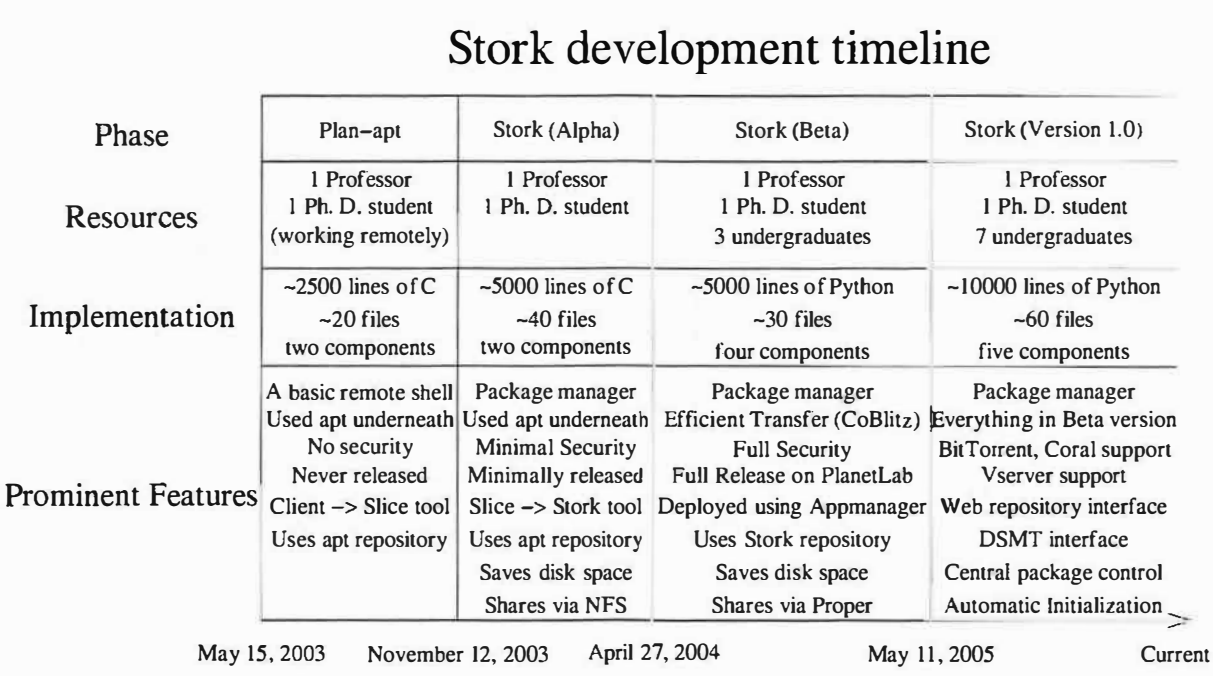


Figure 2: *This timeline shows the evolution of the Stork project. We show the resources used during development, the method of implementation, and the prominent features of each release.*

creating a package manager that efficiently and securely shares content between slices using NFS.

2.1.3 Stork (Beta release)

In the next iteration we rewrote Stork using Python to clean up the code and provide additional functionality missing in the alpha version. We also developed Stork into an independent package management tool no longer reliant on apt. We added support for additional transfer methods and package types. This version of Stork uses packages primarily in the RPM format and resolves dependencies itself. We also use Proper to securely share packages between slices using hard links and file immutable bits. This provides a fast and transparent method to share files instead of NFS.

We divided Stork into four components to handle different tasks. As in the previous version there is a Stork slice on each node as well as a set of installation tools, but the beta version also added repository scripts and a set of tools to authorize package use. Users digitally sign packages and specify to Stork which other users' digital signatures they trust for groups of packages. Stork verifies package signatures before installing a package in a slice.

2.1.4 Stork (Version 1.0)

The latest version of Stork features another re-write and additional functionality. We changed to a more modular design that allows developers to write simple stubs to perform similar actions using different implementations. For example, Stork has a stub interface for network transfers with stubs such as HTTP, CoDeeN, FTP, CoBlitz, and BitTorrent available. A developer could create a new stub for Bullet[9] and then use that stub with Stork without modifying any other code.

2.2 Problems

The problems with different versions of Stork were mainly due to competing interests. Since Stork is a research service, we tried to balance research and usefulness. Where practical, we made use of other research services so as to increase Stork's functionality. For example, Stork downloads content using CoBlitz and BitTorrent, shares files across slices using Proper, and has a novel technique for validating packages. The complexity of providing these features has greatly decreased the usability and stability of Stork as a whole.

When developing the beta version of Stork we decided to include intelligent content transfer. Point-to-point HTTP transfers worked fine for the loads experienced by earlier versions of Stork, but clearly would not scale to larger numbers of users. As a result we added support for other transfer types, including BitTorrent, CoBlitz, and CoDeeN. We allowed the user to choose what transfer type they wanted to use for their transfer, but unfortunately when a transfer type failed, we did not retry with another type. From a research standpoint this was an appropriate simplification because all of the functionality was there. From a service point of view, it caused failures that our users did not appreciate.

Stork also depends on Proper, a research service that enables inter-slice interaction such as file sharing. This dependency has also proven problematic, as getting Stork and Proper to work well together has been difficult. When an error occurs it isn't clear where the problem lies. Stork must depend on Proper because sharing package files between slices is a key motivation for Stork. Sharing files allows Stork to save disk space, avoid unnecessary information downloads, and reduces the memory use of shared programs. However, these features are more beneficial to the PlanetLab infrastructure than individual users; users want a service that always works, rather than one that works most of the time and provides only intangible benefits.

Downloading packages securely is a major feature of Stork. The standard solution is a package repository that is trusted to contain only valid packages. This security model is inappropriate for PlanetLab's unbundled management. Instead, Stork allows users to sign packages digitally, and only install packages with acceptable signatures. Acceptable signatures can be specified on a per-package basis, allowing a user to accept another user's signature for certain packages but not others.

Although these measures greatly increase Stork's security, it makes Stork more complicated for our users. Users must not only sign packages that they upload to our repository, but they must also configure Stork to accept the appropriate signatures. This complexity has been the source of much confusion and frustration by our users.

3 Making Research Services Viable

The requirements for research and stability oppose each other and create a fundamental tension in research service development. This tension helps to create a downward spiral for each research service on PlanetLab and underscores the gap between research and real world practices. In this section we describe the downward spiral, and offer suggestions for escaping it.

3.1 The Downward Spiral of Research Services on PlanetLab

One problem that most services face is they get into a negative cycle that stunts project development and prevents the growth of a strong user base. A service cannot build a user base because it lacks stability and features. The features and stability cannot be provided without a large user base to drive the feature set and do the necessary third-party testing.

Research services on PlanetLab tend to be either very successful in attracting users (e.g. CoDeploy and Coral[7]) or have few users (e.g. Stork, Bellagio[1], and DSMT[6]). There doesn't appear to be any middle ground. Most of the successful research services

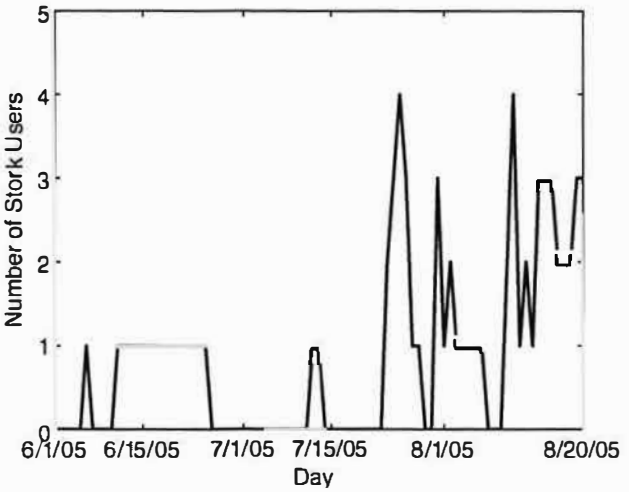


Figure 3: *The number of slices that used Stork on each day from June 1st to August 20th, 2005.*

on PlanetLab are HTTP content distribution networks that provide the same functionality as existing non-PlanetLab services. They do not have dependencies on external research services: either they don't need such services, or the research groups developed the necessary research services themselves. For example, CoBlitz depends on CoDeeN, CoMon, and CoDNS, all developed by the same research group.

The failed research services tend to only be useful for other PlanetLab researchers, and have dependencies on other research services. The former reduces the available user base, while the latter decreases stability. As a research service depends on more and more research services, the size and number of corner cases increases. Eventually, almost the entire operational space is corners, leaving very little of the service functional. This leads users to avoid the service and continues the spiral.

3.2 Escaping the Spiral

We have several suggestions to help improve the quality and usability of research services on PlanetLab.

1. **Fall-back gracefully to independent operation.**
   Although a research service may depend on other research services, it should fall back to more reliable functionality if necessary. For example, Stork uses Proper to share files but will directly download and install them if Proper fails. Stork also falls back to direct HTTP transfers should the more sophisticated download mechanisms fail. In many cases this fall-back does not inconvenience the user, it only decreases the overall efficiency of the system. Since users prefer functionality to efficiency, this seems a reasonable trade-off, at the cost of additional complexity to implement the fall-back mech-

anism.

2. **Build on other research services.**

   The previous suggestion does not mean that research services should never rely on one another. It should just be done in moderation. We believe that inter-dependency is a good thing, as it increases research service functionality and potential user base. To date, we have tried to be active users of any appropriate research service on PlanetLab. Stork uses Proper, CoDeeN, CoBlitz, and AppManager to provide it with improved functionality. We have also investigated using PLuSH[16], DSMT, Coral, Bellagio, PsEPR, and Sirius to various degrees.

3. **There must be a reliable core.**

   While research services are interesting because of the research component, users just want them to work. If a research service provides an operational subset that always works, then users can be assured that there are some functions that are well tested and can be depended upon. In other words, the research service developer should make a reliable service and build the research framework around it.

4. **Incentives are needed for research service creation.**

   Services like AppManager work very well in practice, but are uninteresting from a research standpoint. For that reason few of them exist on PlanetLab, although those that do exist are stable and well-used. The motivation for creating such services isn't clear, but may be similar to that of people who write open source software. Service creators obtain a certain amount of renown within the community but largely donate their effort with the hope that others will also donate useful software.

   If the PlanetLab Consortium were to offer prizes for service creation and deployment (similar to the Ansari X-Prize [20]), it would increase the interest in service creation. Currently there are too many research prototypes and too few services to have a stable base to build upon.

   Perhaps PlanetLab should start its own conferences and/or journals that focus on research services that have real user bases. WORLDS is a step in the right direction, but workshop publications don't have the same weight as conferences and journals. Not only would these publications help other researchers to develop research services, it would give them an incentive to do so. Alternatively, sessions which focus on high-quality experiences papers could be added to existing conferences.

5. **Standardized interfaces are not the solution.**

   The problem with inter-service dependency is not the lack of standardized interfaces. Dealing with different interfaces for different services is not that difficult; adding a new data transfer service to Stork is as simple as writing a few stub routines. The problem is running into corner cases in which the research service does not work correctly. Working around a bug in another service is extremely difficult and time-consuming. Effort should be expended reducing corner cases and documenting those that remain, rather than standardizing interfaces.

4 Conclusion

Throughout this paper we have described the problems that research services face on PlanetLab. The requirements for novelty and interuse cause instability that frustrates users. We have provided suggestions explaining how to build a reliable and stable tool for users without sacrificing the research value of the service. New incentives for research services along with better techniques for building research service would help to develop PlanetLab. With time, PlanetLab may fulfill the dream of having long-running research services running alongside research prototypes.

5 Acknowledgements

We would like to thank Vivek Pai and Steve Muir for suggesting we write this paper and the entire Stork team, especially Jason Hardies, for keeping other projects moving while we worked on this. We would also like to thank our shepherd, Dave Andersen, and the anonymous reviewers for their comments that greatly improved this paper.

References

[1] AuYoung, A., Chun, B., Snoeren, A., Vahdat, A., "Resource allocation in Federated Distributed Computing Infrastructures", In Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure (2004).

[2] Brett, P., Knauerhase, R., Bowman, M., Adams, R., Nataraj, A., Sedayao, J., Spindel, M., "A Shared Global Event Propagation System to Enable Next Generation Distributed Services", First Workshop on Real, Large Distributed Systems (WORLDS), 2004

[3] Brooks, F., "The Mythical Man Month", 1975

[4] CoBlitz, http://codeen.cs.princeton.edu/coblitz/

[5] Cohen, B., "Incentives Build Robustness in BitTorrent", Workshop on Economics of Peer-to-Peer Systems, 2003

[6] Distributed Service Management Toolkit, http://yum.psepr.org/

[7] Freedman, M., Freudenthal, E., and Mazires, D., "Democratizing Content Publication with Coral", In Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04) San Francisco, CA, March 2004.

[8] Huebsch, R., PlanetLab application manager. http://appmanager.berkeley.intel-research.net/

[9] Kostic, D., Rodriguez, A., Albrecht, J., Vahdat, A., "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh", in Proceedings of ACM SOSP, 2003.

[10] Muir, S., Peterson, L., Fiuczynski, M., Cappos, J., Hartman, J., "Proper: Privileged Operations in a Virtualised System Environment", USENIX 2005

[11] Peterson, L., "Dynamic Slice Creation", PDN-02-005 Draft, 2002.

[12] Peterson, L., Anderson, T., Culler, D., Roscoe, T., "A Blueprint for Introducing Disruptive Technology into the Internet", PDN-02-001, 2002.

[13] Peterson, L., Roscoe, T., "PlanetLab Phase 1: Transition to an Isolation Kernel", PDN-02-003, 2002.

[14] PlanetLab Sirius Scheduler, http://snowball.cs.uga.edu/ dkl/pslogin.php

[15] Plkmod. http://www.cs.princeton.edu/ acb/plkmod/

[16] PLuSH, http://sysnet.ucsd.edu/projects/plush/

[17] Stork. http://www.cs.arizona.edu/stork/.

[18] Vservers. http://linux-vserver.org/

[19] Wang, L., Park, K., Pang, R., Pai, V., Peterson, L., "Reliability and Security in the CoDeeN Content Distribution Network", Proceedings of the USENIX 2004 Annual Technical Conference, 2004

[20] XP. http://www.xprizefoundation.com/

Using PlanetLab for Network Research: Myths, Realities, and Best Practices

Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai

1 Introduction

PlanetLab is a research testbed that supports 428 experiments on 276 sites, with 583 nodes in 30 countries. It has lowered the barrier to distributed experimentation in network measurement, peer-to-peer networks, content distribution, resource management, authentication, distributed file systems, and many other areas.

PlanetLab did not become a useful network testbed overnight. It started as little more than a group of Linux machines with a common password file, which scaled poorly and suffered under load. However, PlanetLab was conceived as an *evolvable* system under the direction of a community of researchers. With their help, PlanetLab version 3.0 has since corrected many previous faults through virtualization and substantial performance isolation. This paper is meant to guide those considering developing a network service or experiment on PlanetLab by separating widely-held myths from the realities of service and experiment deployment.

Building and maintaining a testbed for the research community taught us lessons that may shape its continued evolution and may generalize beyond PlanetLab to other systems. First, users do not always search out "best practice" approaches: they expect the straightforward approach to work. Second, users rarely report failed attempts: we learned of the perceived shortcomings described in this paper through conversations, not through messages to the mailing lists. Third, frustration lingers: users hesitate to give another chance to a system that was recently inadequate or difficult to use. These experiences are especially challenging for an evolvable system, which relies on user feedback to evolve so that more users can be supported by features they desire.

We organize the myths in decreasing order of veracity: those that are realities in Section 2, that were once true in Section 3, and those that are false if best practices are employed in Section 4. We summarize the discussion in Section 5.

2 Realities

This section describes widely-cited criticisms of PlanetLab that are entirely true, and are likely to remain so even as PlanetLab evolves.

Reality: Results are not reproducible

PlanetLab was designed to subject network services to real-world conditions, not to provide a controlled environment. By running a service for months or years, researchers should be able to identify trends and understand the performance and reliability their service achieves. An experiment that runs for an hour will reflect only the conditions of the network (and PlanetLab) during that hour.

Various aspects of a service can be meaningfully measured by applying simple rules-of-thumb. Avoid heavily-loaded times and nodes: CoMon [5] tracks and publishes current resource usage on each PlanetLab node. Secure more resources for your experiment from a brokerage service (see Section 3) if needed. Repeat experiments to generate statistically valid results. Finally, regard PlanetLab's ability to exercise a system in unintended ways, producing unexpected results, as a feature, not a bug.

Reality: The network between PlanetLab sites does not represent the Internet

No testbed, no simulator [2], and no emulator is inherently representative of the Internet. The challenges for researchers are to develop experiments that overcome this limitation, perhaps by recruiting real users behind residential access networks, or, failing that, to interpret results taking PlanetLab's special network into account. The challenge for PlanetLab is to evolve so that this limitation is less severe, seeking new sites and new access links.

PlanetLab's network is dominated by global research and education network (GREN) [1] (Internet2 in the United States). However, commercial sites have joined PlanetLab and research sites have connected machines to DSL and cable modem links: 26 sites are purely on the commercial Internet. The question is, how does PlanetLab's network connectivity affect research?

First, some experiments are suitable for the GREN. Claims that a new routing technique can find better routes than BGP are suspect if those better routes take advantage of well-provisioned research networks that are not allowed by BGP policy. However, claims that a service can find the best available route might be accurate even on the GREN: results obtained on the GREN are not necessarily tainted.

Second, services for off-PlanetLab users and network measurement projects that send probes off-PlanetLab observe the commercial Internet. Although most of PlanetLab is on the GREN, most machines also connect to the commercial network or are part of transit ASes. The PlanetFlow auditing service [4] reports that PlanetLab nodes communicate with an average of 565,000 unique IP addresses each day. PlanetSeer [10], which monitors TCP connections between CoDeeN nodes at PlanetLab sites and Web clients/servers throughout the Internet, observed traffic traversing 10,090 ASes, including all tier-1 ISPs, 96% of the tier-2 ISPs, roughly 80% of the tier-3 and 4 ISPs, and even 43% of the tier-5 ISPs. Measurement services like Scriptroute [7] can use the geographic diversity of vantage points provided by PlanetLab to probe the Internet without being limited by the network topology between PlanetLab nodes.

Finally, it is sometimes not the topology of the GREN, but the availability of its very high bandwidths and low contention that calls results into question. Researchers can, however, limit the bandwidth their slices consume to emulate a lower bandwidth link, via user-space mechanisms (e.g., pacing the send rate) or by asking PlanetLab support to lower the slice's outgoing bandwidth cap.

Reality: PlanetLab nodes are not representative of peer-to-peer network nodes

Typically, this is a comment about the high-bandwidth network (see above). Sometimes it means that PlanetLab is a managed infrastructure and not subject to the same churn as desktop systems.

Although PlanetLab is not equivalent to a set of desktop machines—and it is not expected to scale to millions of machines—it can contribute to P2P services. A "seed deployment" on PlanetLab would show the value of a new service and encourage end-users to load the service on desktop machines. End System Multicast [3] instead uses PlanetLab nodes as the "super nodes" of a P2P network. PlanetLab can contribute a core of stable, managed nodes to P2P systems.

3 Myths that are no longer true

Some who tried to use early versions of PlanetLab found challenges that are no longer so daunting because PlanetLab has evolved.

Myth: PlanetLab is too heavily loaded

Although PlanetLab may always be under-provisioned and load is especially high before conference deadlines, this perception is misleading in two ways.

First, upgrades to the OS better tolerate high CPU load, memory consumption, and disk access load. CPU cycles are fairly distributed among slices rather than threads: a slice with 100 threads receives *the same* CPU allocation as a slice with just one. A daemon polices memory consumption, killing slices that use too much when memory pressure is high; users now take greater care in configuring programs that may have a heavy memory footprint to avoid having them killed, which in turn has reduced memory pressure for everyone. Finally, an OS upgrade enables disk access via DMA, rather than programmed I/O, improving performance when the node is swapping.

Second, PlanetLab has two brokerage services, Sirius and Bellagio, that perform admission control to a pool of resources. Researchers can use these services to receive more than a "fair share" of the CPU, for fixed periods of time, during periods of heavy load.

CPU availability measurements. An experiment begun in February 2005 supports the claim that PlanetLab has sufficient CPU capacity. The experiment runs a spin-loop on each PlanetLab node to sample the CPU available to a slice; because of PlanetLab's fair share CPU scheduler, this measurement is more accurate than standard techniques such as the load metric reported by `top`. Figure 1 summarizes seven months of CPU availability measurements. The three lines are the median, 25^{th}, and 10^{th} percentiles of the available CPU across all nodes. The median line shows that most nodes had at least 20% available: a slice on a typical PlanetLab node contends with three to five other slices that are running processes nonstop. The 25^{th} percentile line generally stays above 10%, indicating that fewer than one-fourth of the nodes had less than 10% free. A slice can get nearly 10% of the CPU on almost any node.

CPU time is also available immediately before conference deadlines as well. For example, during the week before the SIGCOMM deadline (February 1–8, 2005), 360 of the 362 running nodes (99%) had at least 10% available CPU, averaged over the week; 328 of the 360 nodes (91%) had at least 20% available. These results show somewhat higher availability than in Figure 1. Some projects may have refrained from using PlanetLab to leave resources available to those running last-minute experiments.

Estimates of available CPU using other metrics are less accurate. In Figure 2, we show the median capacities (a) measured directly using spin loops, (b) estimated using the inverse of the load average (a load of 100 equals 1% CPU availability), and (c) estimated using the inverse of the number of active slices (meaning slices with a runnable thread). The top line, the spin-loop measured capacity, is significantly higher. The Unix-reported load average is often misleading: the processors did have high load (sometimes exceeding 100), but the CPU available to slices is much greater because although slices that spawn many processes increase the load average, their processes compete only against each other for CPU. Likewise, not all active slices use their entire quanta and so the active slice count overestimates contention. The CoMon monitoring service now publishes the results of the spin-loop tests to help users choose nodes by CPU availability.

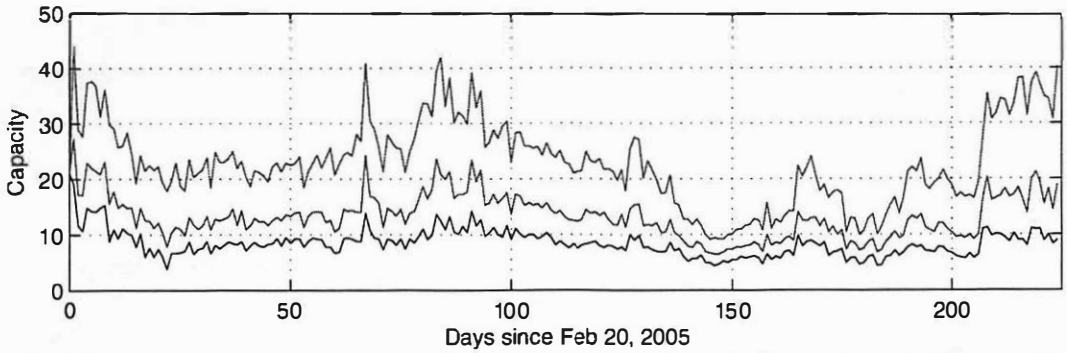


Figure 1: Available CPU across PlanetLab nodes. Median percentage available CPU is red (upper), 25^{th} percentile is green (middle), and 10^{th} is blue (lower).

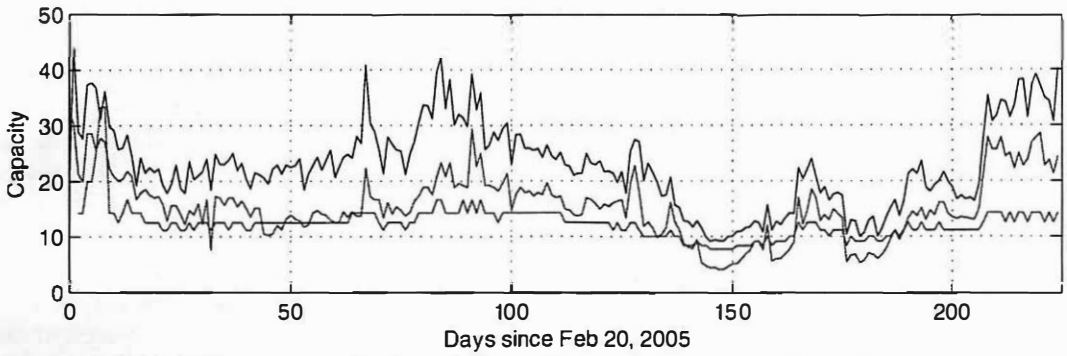


Figure 2: Median available CPU measurements using spin loops (blue, upper), load average (green, middle), and number of active slices (red, typically lowest).

Myth: PlanetLab cannot guarantee resources

Resource guarantees could not be given before version 3.0. Schedulers are now available to make resource guarantees, but PlanetLab does not yet have a policy about what slices should receive them. Typically, continuously running services on PlanetLab are robust to varying resource availability (and have not asked for guarantees), while short-term experiments have the option of using one of the brokerage services (see previous item) to gain sufficient capacity for the duration of a run. Once we have enough experience to understand what policies should be associated with guarantees, or someone develops a robust market in which users can acquire resources, resource guarantees are likely to become commonplace.

4 Myths falsified by best practices

The following four myths about PlanetLab are not true if best practices are followed. Often these myths are caused by mismatches between the behavior of a single, unloaded Linux workstation, and the behavior of a highly-shared, network of PlanetLab-modified Linux nodes. The first three myths address problems using PlanetLab for network measurement, the last, its potential for churn.

Myth: Load prevents accurate latency measurement

Because PlanetLab machines are loaded, no application can expect that a call to `gettimeofday()` right after `recv()` will return the time when the packet was received by the machine. The PlanetLab kernel scheduler (Section 3) can isolate slices so that none are starved of CPU, but cannot ensure that any slice will be scheduled immediately upon receiving a packet.

Using in-kernel timestamping features of Linux, network delay can be isolated from (most) processing delay. When a machine receives a packet, the network device sends an interrupt to the processor so that the kernel can pull the packet from the device's queue. At the point when Linux accepts the packet from the device driver, it annotates the buffer with the current time.[1] The kernel will return control to the current process for the remainder of its quantum, but this timestamp is kept in the kernel and made available in at least three ways:

1. The SIOCGSTAMP ioctl called after reading a packet. Ping uses this ioctl, but Linux kernel comments suggest the call is Linux-specific.
2. The SO_TIMESTAMP socket option combined with `recvmsg()`: ancillary data includes a timestamp. The Spruce [8] receiver code uses this method, which was introduced in BSD and is supported by Linux. It is not widely documented, but can be run as a non-root user.
3. The library behind tcpdump, libpcap. This may be the most portable, but requires root, which is easy on PlanetLab. *Sent* packets are also timestamped [9].

[1] See: linux/net/core/dev.c:netif_rx().

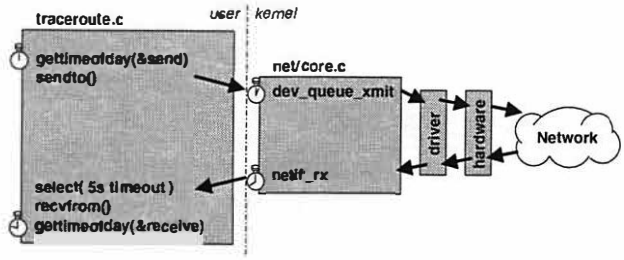


Figure 3: Approaches to packet round-trip timing: applications can use gettimeofday before sending and after receiving; closer to the device are kernel-supplied timestamps applied as the packet is queued for transmission or received. The driver and hardware also may delay packets on transmission and receipt.

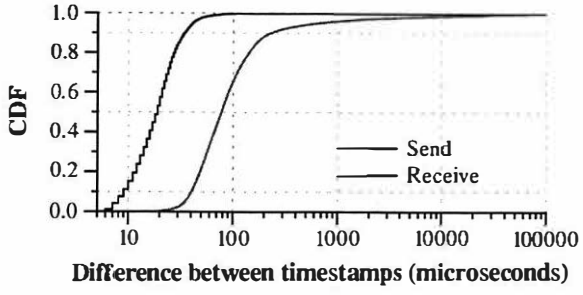


Figure 4: A cumulative distribution of the differences between application-level timestamps and kernel-level timestamps when sending (left) or receiving (right) in microseconds.

Do kernel timestamps matter? To collect samples of application- and kernel-level timestamps, we modified traceroute to print the timestamps it collects via `gettimeofday()`, then ran traceroute and tcpdump in parallel to gain kernel-level timestamps for the same packets from 300 PlanetLab machines to three destinations, collecting 40,000 samples for comparison. Figure 3 illustrates where traceroute and the kernel annotate timestamps.

In Figure 4, we show the differences between application- and kernel-captured timestamps when sending probes and receiving responses. Although the time between `gettimeofday()` and when the packet is delivered to the network device is typically small (18 μs median, 84 μs mean), the time after the packet is received is typically larger and more variable (77 μs median, 788 μs mean). The larger median may represent the cost of the intermediate system calls: in traceroute, it is `select()` that returns when the response packet is received. However, that 4% of samples are above 1 ms suggests contention with other active processes. Further, the smallest 3% of samples between 20–30 μs suggests that tools that filter for the minimum round trip time, such as pathchar, will have difficulty: 97% of the packets will not observe minimal delay in receive processing.

Measurement tools downloaded from research Web pages may not use kernel-level techniques to measure packet timings; their results should be held with skepticism until their methods are understood.

Myth: Load prevents sending precise packet trains

Sending packets at precise times, as needed by several tools that measure available bandwidth, is more difficult. If the process is willing to discard measurements where the desired sending times were not achieved or when control of the processor is lost, then sending rate-paced data on PlanetLab simply requires more attempts than on unloaded systems.

To determine how CPU load impairs precise sending, we measure how often we can send precisely-spaced packets in a train. Sent trains consist of eleven packets, spaced either by 1 ms, to test spin-waiting, or 11 ms, to test sleep-based waiting using the `nanosleep()` system call (via the `usleep()` library call). We show how often the desired gaps were achieved for 1 ms gaps in Figure 5 and 11 ms gaps in Figure 6. In all measurements, 10 gaps are used, and we measure how often the gaps are within 3% of the target either for all 10 gaps or for any 5 consecutive gaps.

For both tests, at least five consecutive gaps have the desired intervals in 80–90% of the trains. For the 11 ms test, all 10 gaps had the correct timing 60–70% of the time. The 1 ms test did not fare as well: all 10 gaps met their target times in only 20–40% of the trains. For the shorter (5-gap) chirp trains, the results are quite good: sending 10 packets is sufficient to discard less than 20% of the measurements. For longer chirp trains, two to five times as many probes may have to be sent, which may be tolerable for many experiments.

Mechanisms for negotiating temporarily longer time slices, or even delegating packet transmission scheduling to the kernel, are being discussed. The latter might address another source of concern for measurement experiments: the packet scheduler used to cap bandwidth and fairly share bandwidth among slices. The timestamps on sent packets that a process can observe with libpcap are accurate—the kernel timestamps packets *after* they pass through the packet scheduler—and so can still be used to discard bad results. However, the scheduler does limit the kinds of trains that can be sent: it enforces a per-slice cap of 10 Mbps with a maximum burst size of 30KB. Longer trains sent at a faster rate are not permitted.

Myth: The PlanetLab AUP makes it unsuitable for measurement

The PlanetLab user Acceptable Use Policy [6] states:

> PlanetLab is designed to support network measurement experiments that purposely probe the Internet. However, we expect all users to adhere to widely-accepted standards of network etiquette in an effort to minimize complaints from network administrators. Activities that have been interpreted as worm and denial-of-service attacks in the past (and should be avoided) include sending SYN packets to port 80 on random machines, probing random IP addresses, repeatedly pinging routers, overloading bottleneck links

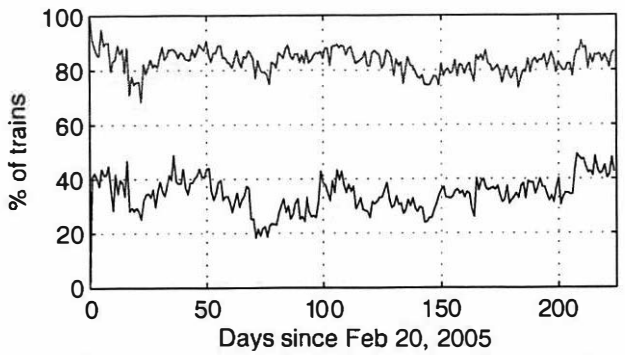


Figure 5: Timing statistics for 1 ms (spin-based) chirp trains. The green (upper) line indicates at least 5 consecutive gaps met the target timings, while the blue (lower) line indicates all gaps met the target.

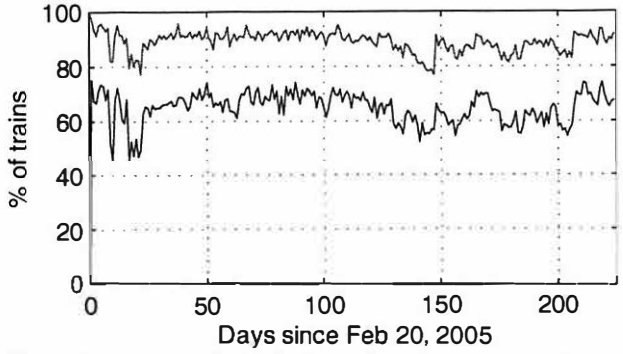


Figure 6: Timing statistics for 11 ms (sleep-based) chirp trains. The green (upper) line indicates at least 5 consecutive gaps met the target timings, while the blue (lower) line indicates all gaps met the target.

with measurement traffic, and probing a single target machine from many PlanetLab nodes.

This policy is a result of experience with network measurements on PlanetLab, and is designed to prevent network abuse reports of the form "PlanetLab is attacking my machine." Here we elaborate on steps to conduct responsible Internet measurement on PlanetLab. The goal of these practices is to make network measurements as easy to support as possible by building a list of hosts that "opt-out" of measurement without growing the list of PlanetLab sites that have asked to "opt-out" of hosting measurement experiments.

Test locally and start slow. Do not use PlanetLab to send traffic you would not send from your workstation. Use a machine at your site first to discover any problems with your tool before causing network-wide disruption. Measurements from PlanetLab can appear to be a distributed denial of service attack; starting with a few nodes can limit how many sites receive abuse reports. Some intrusion detection systems generate automatic abuse reports; an abuse report to every PlanetLab host is best avoided.

Software has bugs, and bugs can cause measurements to be more intrusive than necessary. Bugs that have made PlanetLab-supported tools unnecessarily intrusive include faulty checksum computation in a lightweight traceroute implementation and a reaction to unreachable hosts that directed a great deal of redundant measurement toward the same router. Such errors could have been detected before deployment with local testing.

Even a correctly-implemented tool may require local testing, because very little experimental data guides non-intrusive measurement tool design: are TCP ACKs less likely to raise alarms than SYNs? Should traceroute not increment the UDP destination port to avoid appearing as a port scan? How many probes are needed to distinguish lossy links from unreachable hosts?

Starting slow could have avoided abuse report flurries in March and October 2005. An experiment with an implementation flaw generated 19 abuse reports from as many sites, half on the first day, March 15. The experiment ran for only 21 hours before being shut down, but reports continued in for two weeks. A carefully-designed experiment in October tickled two remote firewalls and a local intrusion detection system for a total of 10 abuse reports forwarded to PlanetLab support. The automated responses from remote firewalls may have been avoided by local testing of the destination address list. Many more abuse reports were likely generated by the automated systems, but discarded by recipients as frivolous as they reported a single ICMP echo request (ping) as an attack.

Alert PlanetLab support. Update your slice description *and* send a message to PlanetLab support detailing your intended measurement, how to identify its traffic, and what you've done to try to avoid problems. First, sending such a message shows that you, as an experimenter, believe you have put sufficient effort into avoiding abuse reports. Second, describing your approach gives PlanetLab staff and other interested people the chance to comment upon your design. Finally, knowing the research goals and methods can save PlanetLab staff time and ensures prompt response to abuse reports.

Use Scriptroute. Scriptroute separates measurement logic from low-level details of measurement execution. It will prevent contacting hosts that have complained about traffic, can prevent inadvertently invalid packets that trigger intrusion detection systems, will limit the rate of traffic sent, collects timestamps from libpcap, and schedules probes using a hybrid between sleeping and busy-waiting.

Curtail ambition. It is tempting to demonstrate implementation skill by running a measurement study from *everywhere* to *everywhere*, using many packets for accuracy, and using TCP SYN packets to increase the chance of discovering properties of networks behind firewalls. Resist! Aggressive measurement increases its cost for only a marginal benefit to the authority of your result.

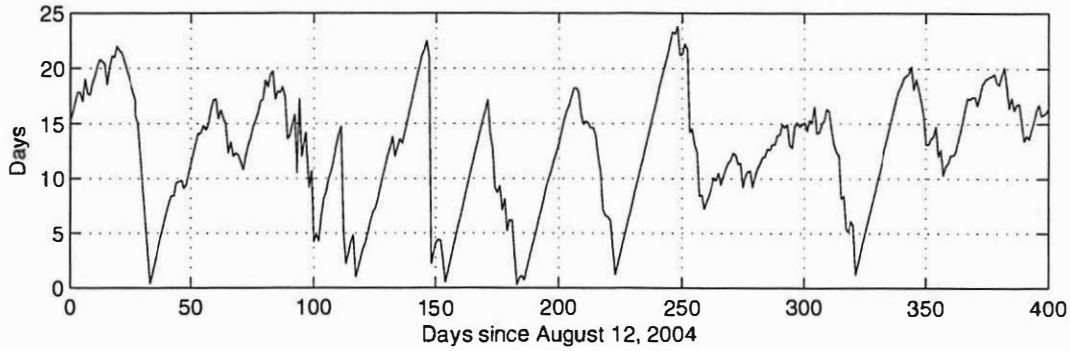


Figure 7: Median uptime in days across all PlanetLab nodes.

Myth: PlanetLab experiences excessive churn

Widespread outages on PlanetLab are fairly rare. Only three times during the last two years have many PlanetLab nodes been down for longer than a reboot: (1) all nodes were taken off-line for a week in response to a security incident in December 2003; the system was also upgraded from version 1.0 to 2.0; (2) an upgrade from version 2.0 to 3.0 during November 2004 caused more churn than usual for a two week period; and (3) a kernel bug in February 2005 took many nodes off-line for a weekend.

On the other hand, roughly 30% of PlanetLab's nodes are down at any given time. About one-third of these are down for several weeks, usually because a site is upgrading the hardware or blocking access due to an AUP or security issue. The remaining failed nodes are part of the daily churn that typically sees 15–20 nodes fail and as many recover each day. Major software upgrades that require reboots of all nodes occur, but are infrequent.

PlanetLab as a whole has been remarkably stable. Figure 7 shows median node uptimes over 13 months. Of the six sharp drops in uptime, four are due to testbed-wide software upgrades requiring reboots. The longer upgrade, to version 3.0, is shown starting at day 100. The kernel bug, followed by an upgrade, is evident starting at day 170. Median uptimes are generally longer than 5 days, and often 15 to 20 days—much higher than what would be expected in typical home systems.

Since PlanetLab does experience churn, no users should expect that the storage offered by PlanetLab nodes is persistent and no users should expect that a set of machines, once chosen, will remain operational for the duration of a long-running experiment.

5 Summary

In this paper, we described realities of the PlanetLab platform: it is not representative of the Internet or of peer-to-peer networks, and results are not always reproducible. We then described myths that linger despite being fixed: PlanetLab's notoriously high load poses less of a problem today than it once did because there are resource brokerage services and the operating system has been upgraded to isolate experiments. Finally, we described challenges that can often be addressed by following some best practices. PlanetLab is capable of substantial network measurement, despite technical challenges in precise timing and social challenges in avoiding abuse complaints. In addition, many PlanetLab machines may fail or be down at any time; being prepared for this churn is a challenge for experimenters.

Our hope is that separating myth from reality will make clear the features and flaws of PlanetLab as an evolving research platform, enabling researchers to choose the right platform for their experiments and warning them of the challenges PlanetLab implies.

Acknowledgments

We would like to thank the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0435065.

References

[1] S. Banerjee, T. G. Griffin, and M. Pias. The interdomain connectivity of PlanetLab nodes. In *PAM*, 2004.

[2] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, 2001.

[3] Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS*, 2000.

[4] M. Huang, A. Bavier, and L. Peterson. PlanetFlow: Maintaining Accountability for Network Services. Submitted for publication.

[5] K. Park and V. Pai. CoMon: A monitoring infrastructure for PlanetLab. http://comon.cs.princeton.edu.

[6] PlanetLab Consortium. PlanetLab acceptable use policy (AUP). https://www.planet-lab.org/php/aup/PlanetLab_AUP.pdf, 2004.

[7] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *USITS*, 2003.

[8] J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *IMC*, 2003.

[9] TCPDUMP.org Frequently Asked Questions. http://www.tcpdump.org/faq.html, 2001.

[10] M. Zhang, *et al.* PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *OSDI*, 2004.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications; see http://www.usenix.org/membership/specialdisc.html

SAGE, The System Administrators Guild

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

For more information about membership, conferences, or publications, see http://www.usenix.org.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

Addison-Wesley Professional/Prentice Hall Professional • AMD • Asian Development Bank
Cambridge Computer Services, Inc. • EAGLE Software, Inc. • Electronic Frontier Foundation
Eli Research • GroundWork Open Source Solutions • Hewlett-Packard • IBM • Intel • Interhack
The Measurement Factory • Microsoft Research • NetApp • Oracle • OSDL • Perfect Order
Raytheon • Ripe NCC • Sendmail, Inc. • Splunk • Sun Microsystems, Inc.
Taos • Tellme Networks • UUNET Technologies, Inc.

SAGE Supporting Members

Asian Development Bank • FOTO SEARCH Stock Footage and Stock Photography
Microsoft Research • MSB Associates • Raytheon • Splunk • Taos • Tellme Networks

ISBN 1-931971-40-4